MASTER EN SCIENCES INFORMATIQUES

Sampling & Testing all configurations: The JHipster case study

Halin, Axel; Nuttinck, Alexandre

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculty of Computer Science
Academic Year 2016–2017

Sampling & Testing all configurations:

The JHipster case study

Axel Halin Alexandre Nuttinck

Internship mentor: Acher Mathieu

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
Heymans Patrick

Co-supervisors: Perrouin Gilles
Devroey Xavier

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

Many approaches for testing configurable software systems start from the same assumption: it is impossible to test all configurations. This motivated the definition of variability-aware abstractions and sampling techniques to cope with large configuration spaces. Yet, there is no theoretical barrier that prevents the exhaustive testing of all configurations by simply enumerating them, if the effort required to do so remains acceptable. Not only this: we believe there is lots to be learned by systematically and exhaustively testing a configurable system. We report on our endeavor to test all possible configurations of an industry-strength, open source configurable software system, JHipster, a popular code generator for web applications. We built a testing scaffold for the 26,000+ configurations of JHipster using a cluster of 80 machines for a total of 4376 hour-machine. We find that 34.37% configurations fail and we identify the feature interactions that cause the errors. We show that sampling testing strategies (like dissimilarity and 2-wise) (1) are more effective to find faults than the 12 default configurations used in the JHipster continuous integration; (2) can be too costly and exceed the available testing budget. We cross this quantitative analysis with the qualitative assessment of JHipster's lead developers.

Key words: *Case Study*; *Web-apps*; *Variability-related Analyses*; *JHipster*; *Software Product Line*; *Software Product Line Testing*; *t-wise criteria*;

De nombreuses approches de test de systèmes logiciels configurables partent du même postulat: il est impossible de tester toutes les configurations. Cette hypothèse a motivé la définition de techniques d'échantillonnage et d'abstraction de la variabilité pour faire face aux grands espaces de configuration. Il n'existe cependant pas de barrière théorique empêchant le test exhaustif de toutes les configurations en les énumérant, pour autant que l'effort requis reste raisonnable. Nous pensons d'ailleurs que beaucoup peut être appris d'un test exhaustif d'un système configurable. Nous présentons notre effort pour tester toutes les configurations de JHipster, un générateur d'applications Web. Nous avons construit une infrastructure de test pour ses +26.000 configurations en utilisant un cluster de 80 machines pour un total de 4376 heure-machine. Nous identifions les interactions de fonctionnalités provoquant l'échec de 34.37% des configurations. Nous montrons que les stratégies d'échantillonnage (dissimilarité, 2-wise) sont (1) plus efficaces pour trouver des fautes que les 12 configurations utilisées dans l'intégration continue de JHipster; (2) trop coûteuses et dépasser le budget de test. Nous croisons cette analyse quantitative avec l'évaluation qualitative des développeurs principaux de JHipster.

Mots clés: *Cas d'étude*; *Applications Web*; *Analyses de la varabilité logicielle*; *JHipster*; *Lignes de produits logiciels*; *Test de Lignes de Produits Logiciels*; *Test combinatoire d'interaction*;

Preface

This master thesis is written by Axel Halin and Alexandre Nuttinck as part of their last year in Computer Sciences at the University of Namur. This thesis is the product of the master period including an internship at INRIA Rennes-Bretagne Atlantique, in the DiverSE team, under the supervision of our promoters Patrick Heymans, Gilles Perrouin and Xavier Devroey and our internship mentor Mathieu Acher (Rennes 1) from September 15th 2016 to January 15th 2017.

First we would like to highlight and thank the remarkable supervision of both our supervisors (Gilles, Xavier and Patrick) and our internship mentor (Mathieu). Thank you for an interesting and pleasant experience !

We also thank the DiverSE team for their warm welcome and the numerous discussions around a nice cup of coffee.

Trugarez, Kenavo ar wech all !

Finally, we would like to thank everyone who participated to the study presented in this thesis, either by their constructive comments or reviews. Thank you all.

Halin Axel and Nuttinck Alexandre

I would like personally to thank my friends and family to keep me motivated during all these years studying computer science at the University of Namur. I would also like to thank you Bernard for your constructive comments to the thesis. Thank you Eleonore to always be there for me.

Alexandre

I would like to thank my family and friends for their tremendous support throughout my years in Namur. More specifically, I would like to thank my sister, Stéphanie, for her many reviews and advices and Hélène for being there for me.

Axel

Contents

List of Figures

List of Tables

Part I.

Introduction

Glossary

- **Defect**: A *defect* refers to either a fault or a failure.
- **DSL** (**D**omain **S**pecific **L**anguage): "DSL provides a notation tailored towards an application domain and is based on the relevant concepts and features of that domain. As such, a DSL is a means to describe and generate members of a family of programs in the domain." (Van Deursen & Klint, 2002)
- **Failure**: A *failure* is an "undesired effect observed in the system's delivered service" (Mathur, 2008; IEEE Computer Society, 2014) (e.g., the JHipster configuration fails to compile).
- **Fault**: We consider that a *fault* is a cause of failures. (e.g., as we found in our experiments – see Section 6.3 – a single fault can explain many configuration failures since the same feature interactions cause the failure.)
- **Feature**: A *feature* is a product characteristics that the product has or delivers (Kyo et al., 2002). It is explained with more details in Section 1.1.
- **FM** (**F**eature **M**odel): *FM* is a tree-like graph where nodes are features with relationships among them (constraints). It is used to represent the variability of *SPL*. It is presented in more details in Section 1.2.1.
- **Glue (code)**: Specific code required to generate/compile/run a specific configuration. It can be a script starting a database service or the code needed to deploy a configuration with Docker for instance.
- **SPLE** (**S**oftware **P**roduct **L**ine **E**ngineering): Software engineering paradigm aiming at mass producing software with possibility of tailoring it to the needs of many users. The concept is explained in Chapter 1.

Introduction

Inspired from David Parnas' programs families (Parnas, 1976) and borrowing ideas from the Ford's product line, the Software Product Lines (SPLs) paradigm was popularized in the 2000s. It aims at achieving mass customization, i.e. the mass production of software while offering the capability to tailor it to the needs of many different users. With this paradigm rose the need of (highly) configurable systems, offering numerous options (or features) that promise to fit the needs of different users.

With configurable systems, new features can be activated or deactivated while some technologies can be replaced by others to address a diversity of deployment contexts, usages, etc. The engineering of highly configurable systems is a standing goal of numerous software projects but it also has a significant cost in terms of development, maintenance and testing.

Moreover, Software Product Lines have contributed to the spreading variability-management ideas everywhere. These variability-intensive systems (or *highly configurable systems*) include, among others, web systems such as Drupal and Wordpress.

A major challenge for developers of configurable systems is to ensure that all combinations of options (*configurations*) correctly compile, build and run. The Linux kernel, for instance, offers more than 14000 options (Gazzillo, 2015): how can we ensure that all those configurations are correct? This correctness is even more critical when failing configurations can impact system users, lead to missing opportunities and hinders the success or reputation of a project.

Different testing approaches focus on this problem. Formal methods and program analysis, for instance, which lead to variability-aware testing approaches. However, a common practice is still to execute and test a sample of representative variants, enumerating all configurations being perceived as impossible, unpractical or both - it is especially true with the Linux example mentioned here-before. Despite this generally accepted idea, we are convinced that much has to be learned from exhaustively testing a configurable system. Indeed, knowing all failures of the configurable system would allow the assessment of the error-detection capabilities of sampling techniques with a ground truth.

The research community is facing yet another issue: the lack of case studies. Historical Software Product Lines contains industrial secrets their owners do not want to disclose to a wide audience. The open source community, on the contrary, has contributed to large-scale cases such as Eclipse or Linux.

To tackle this lack of case studies, we introduce *JHipster*, an open source project started in October 2013. JHipster is a popular code generator for web applications. It offers the generation of complete technological stacks composed of Java and Spring Boot code on the server side and Angular and Bootstrap on the front-end side.

JHipster has been found to be an interesting case study for several reasons. First, it relies on a variety of languages and advanced technologies to scaffold web applications and so, introduce variability at different levels. Second, JHipster offers 48 configurations options and 15 constraints between options, leading to more than 150.000 configurations, thus making it manageable but still of an industrial size. In comparison, Linux Kernel 2.6.28.6, for instance, is composed of 5426 features (She et al., 2010) and up to 7000 options in later versions (Apel et al., 2013). Finally, the variability is scattered across numerous kinds of artefacts.

In this study, we aim at exhaustively testing JHipster configurations. To this end, we developed a complete infrastructure including an automated derivation and testing process, that we ran on all +26,000 configurations determined by a refinement of the variability model. That being done, we were able to characterize the cost of such an infrastructure.

We quantified both the engineering effort required to develop the process but also the resources necessary to achieve the execution on all configurations.

In this work, we solely focused on functional properties of JHipster configurations (i.e, do they generate, compile and build successfully?). Non-functional properties are outside the scope of this study and are left for future work.

In this master thesis, we will present the results obtained in our all-configurations testing effort and compare these results with some state of the art sampling techniques (such as t-wise criteria or dissimilarity techniques). We found for instance that about 34.37% of all configurations fail to build and that these failures were caused by 6 interactions of features.

We also initiated the dialogue with the core developers of the project and formulated recommendations regarding their testing strategies. The limited testing budget leads them to test only 12 configurations (19 in the latest versions of the generator) selected on the basis of most-used features. For example, MySQL-based variants are more representatives than Oracle-based variants, while Maven is more represented in the sample than Gradle. We summarize our exchanges and present our recommendations for the JHipster team.

Contributions

The main contributions and findings of this master thesis are:

- an empirical study of a configurable system based on an exhaustive testing;
- a cost assessment and qualitative insights of engineering an infrastructure able to automatically test all configurations. This infrastructure relies on a different configuration-specific *bash* scripts to initiate services, start databases and so on. This infrastructure is itself a configurable system and requires a substantial, error-prone, and iterative effort (8 man-month);
- a computational cost assessment of testing all configurations using a cluster of distributed machines. Despite some optimizations, 4376 hour-machine and 5,2 Tb are needed to execute 26000+ configurations;
- a quantitative and qualitative analysis of failures and faults. We found that 34.37% configurations fail – they either do not compile, build or run. 6 feature interactions (up to 4-wise) mostly explain this high percentage;
- an assessment of sampling techniques. Dissimilarity and t-wise sampling techniques are effective to find faults that cause a lot of failures;
- a retrospective analysis of JHipster practice. The 12 configurations used in the continuous integration for testing JHipster were not able to find the defects. It takes several weeks for the community to discover and fix the 6 faults;
- a discussion on the future of JHipster testing based on collected evidence and feedback from JHipster's lead developers.

Part II.

State of the art

1. Software product lines and variability

To contextualize this thesis and its objectives, we first introduce key notions used in later parts. First, we present in Section 1.1 the notion of *Software Product Lines* (SPLs), their origins and their goals. We then present in Section 1.2 an important notion in SPLs: *variability*. We briefly summarize what it is and present some techniques to both model it and concretely implement it. We conclude this Chapter, by presenting in Section 1.3 *variability-intensive systems*, a generalization of SPLs.

1.1. Software Product Line

Software Product Lines take their origins in Ford's product line (Pohl et al., 2005). If the latter came from the increasing number of buyers for various products, and with that the need of a new way to produce for mass market more cheaply and more rapidly, the former follows the same trend. Indeed, new software products are developed constantly and they are often quite similar. In the transport industry, for instance, even if two companies do not offer exactly the same services (one could provide live tracking of its packages, for example), they might still have the same backbone (account creation and management, online forms, and so on).

These customized products, tailored to the specific needs of the stakeholders, lead to the apparition of a new paradigm: *Software Product Line Engineering* (SPLE). This new paradigm aims at producing software products, using all kinds of reusable artefacts and technological capabilities and mass customization (Pohl et al., 2005).

Software Product Line Engineering is composed of two main activities: *domain engineering* and *application engineering*. The former consists in developing core assets to be configured and combined in order to create different products of the product line, while the latter consists in deciding, in a configuration process, which assets are selected for inclusion and which are discarded (Hubaux et al., 2009). This framework is presented in Figure 1.1.

In their framework, Pohl *et al.* define the two activities as follows (Pohl et al., 2005):

Domain Engineering
The commonality and the variability of the product line, as well as its scope (Pohl et al., 2005) are defined during this process. It also aims at defining and constructing the different artefacts to be reused in the different products.

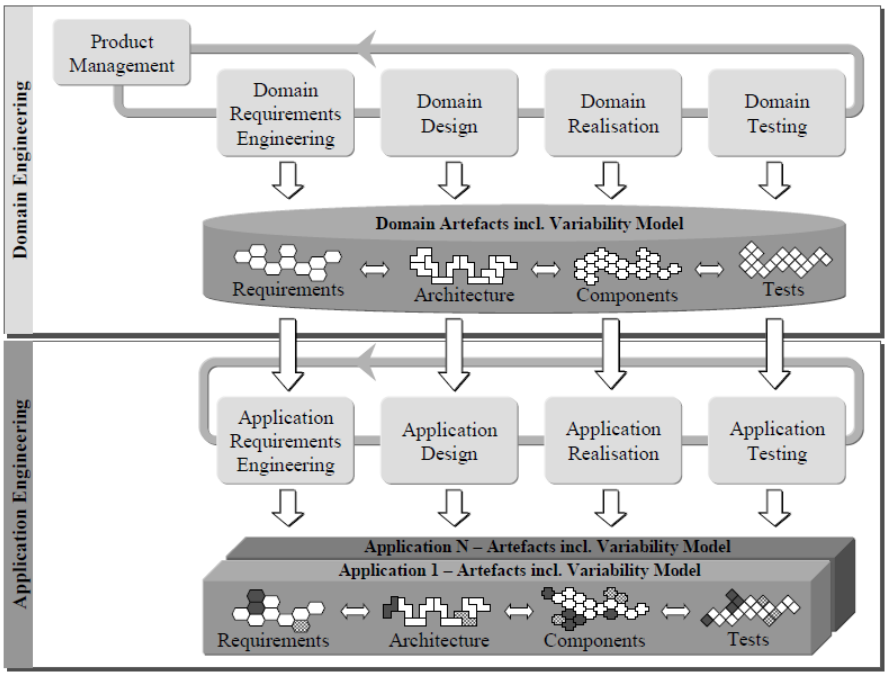

Figure 1.1.: Software Product Line Engineering framework, retrieved from (Pohl *et al.*, 2005)

This process is further divided in 5 sub-processes. Each of them refines the variability and provides feedback to preceding sub-processes on the possible realization of the variability (Pohl et al., 2005):

- *Product Management*: defines the scope of the product line (economic aspects).
- *Domain Requirements Engineering*: elicits and documents common and variable requirements.
- *Domain Design*: defines a common structure for all configurations of the product line.
- *Domain Realization*: designs and implements reusable software artefacts.
- *Domain Testing*: validates the different reusable components.

Application Engineering
During this process, product line applications are derived from the reusable artefacts defined in the domain engineering process (Pohl et al., 2006). Some key goals have been identified by (Pohl et al., 2005):

- Reuse, if it's possible, a maximum of the domain assets when defining and developing a product line application.

- Exploit the variability of the SPL during the development of a product line application.
- Document the application artefacts (e.g. application components) and relate them to the domain artefacts.
- Bind the variability according to the application needs.
- Estimate the impacts of the differences between *application* and *domain*.

Software Product Lines (SPLs) are then used to create large number of related products by reusing a set of software artefacts (Siegmund et al., 2008), thus reducing development effort and time-to-market and providing a high degree of reuse (Hetrick et al., 2006).

The products of a SPL are defined and distinguished in terms of the features they provide. The combination of these features allow to obtain the final software products. As defined by (Kang et al., 1990), a "feature is a prominent or distinctive user-visible behavior, aspect, quality, or characteristic of a software system". Features define both common aspects of the *domain engineering* as well as differences between related systems in the domain. Features can also be used to define the domain in terms of the mandatory, optional, or alternative characteristics of these related systems. One challenge for SPL engineering is to ensure that all products meet their specifications without having to test each individual product, by checking the product line itself.

Moreover, feature interaction is a well-known problem (Zave, 1993)It is quite complex to ensure that all features will interact only in desired ways. Adding or removing a feature to a system may also have an impact on the others (Van Gurp et al., 2001).

To illustrate SPLs, let us consider a trivial example : the vending machine case (Classen et al., 2011). In this example, we consider a manufacturer that provides vending machines to different places. These different customers can have different needs in term of the machine they offer to their own clients. For instance, one provides credit-card payment option while the others limit themselves to cash payments. Moreover, one allows the possibility to pay with different currencies. However, one can concede that they all have the same basic functionality: serve soda. Moreover, each vending machine must run software adapted to the selected set of hardware features. The manufacturer will then have some artefacts common to all products and some specific ones. He will then develop a Software Product Line. He will still be confronted, however, to a common challenge: how to ensure that all products are valid? This challenge is addressed in Chapter 2.

Although many companies (NASA, General Motors, etc.) rely on Software Product Line Engineering (Thüm et al., 2012), very few real cases are openly available. This is mostly due to the industrial secrets they contain and that their owners do not wish to disclose. We can, nonetheless, find large-scale open source cases such as Eclipse (Greiler et al., 2012a), Linux Kernels (She et al., 2010; Abal et al., 2014) or Web-based plugins such as

Drupal[1] (Sánchez et al., 2017) and Wordpress[2] (Nguyen et al., 2014a). This variety of case-studies offered by the open-source community also motivated our decision to work with JHipster as presented in Chapter 3.

We know present a key aspect of the software product lines: the notion of *variability*.

1.2. Variability

As we previously mentioned, SPLE aims at developing software using mass customization. In order to do so, it proposes to reuse artefacts (code, etc.) and to assemble them in a new product. This is achieved through the identification and the management of commonalities and variations in a set of system's artefacts (Chen et al., 2009).

From this vision appears *variability*. Many definitions have been advanced for this notion, ranging from *"the ability to change or customize a system"* (Van Gurp et al., 2001) to *"the ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion"* (Bachmann & Clements, 2005). Pohl *et al.* define variability of a SPL as: *"variability that is modelled to enable the development of customised applications by reusing predefined, adjustable artefacts"* (Pohl et al., 2005).

Furthermore, Pohl *et al.* distinguish *internal variability* from *external variability*. The former is hidden from the customers and it often arises from a refinement of the latter. For instance, it can be two types of communication protocol, each having their own advantages and drawbacks. The latter, is visible to the customer and directly impacts his satisfaction. It can be, for example, the type of authentication used (keypad, facial recognition and so on).

From these different definitions, we can gather that variability is *the ability of a system or an asset to be customized through the use of modifiable artefacts sometimes called variation points, or features in feature-based approaches*. Variability can then be seen at two granularity levels: the variability of the system, which is the set of reusable artefacts and the variability of an artefact (code, *etc.*) having common and specific parts. In the case of code artefacts, for instance, the variability can be the instructions shared by all instances and instructions specific to some, and which will not be executed in other instances (see Section 1.2.2 for some implementation mechanisms).

Moreover, and as introduced in the previous Section, SPLE can be seen as a two-step process: domain engineering and application engineering. In this paradigm, variation points are introduced during domain engineering while variability decreases in application engineering through the binding of the variation points (the selection of the artefacts, ...)

[1] `https://www.drupal.org/`
[2] `www.wordpress.com/`

(Bosch et al., 2002). Moreover, this binding can be achieved at different times (Gacek & Anastasopoules, 2001):

- **Compile-time**: before the actual compilation of the program (through the use of preprocessor directives, for instance) or at compile time;
- **Link-time:** during module or library linking (for example, selecting different libraries with different versions);
- **Runtime**: during program execution (depending on some predicates – such as privileges level for instance – some functionalities can be enabled or disabled);
- **Update-time or post-runtime**: during updates of the program or after its execution (for instance, adding new functionalities to a module through an update utility).

As noted by Pohl *et al.* , variability information must be available when deciding on the commonality and variability of the product line (*domain engineering*) and when the variability is bound to develop individual products (*application engineering*) (Pohl et al., 2006). As such, the management of the variability is an essential activity to the success of a configurable system. As (Schmid & John, 2004) report, it is also "*the* key feature that distinguishes Product Line Engineering from other approaches to software development".

1.2.1. Variability management

Variability management regroups many activities – the explicit representation of the variability in software artefacts, the management of dependencies and the support of instantiations of those variabilities (Chen et al., 2009) – to document the product line variability (Pohl et al., 2006).

Moreover, (Schmid & John, 2004) identified 5 key challenges variability management ought to tackle: the definition of variation points and elements potentially bound to them; the definition of relations and constraints among variation points and a selection mechanism to define the elements that should go into a specific product.

This variability management, however, is far from trivial and is concerned with many issues, at different phases of the life cycle as reported by (Bosch et al., 2002). In their article, the authors identify several issues, either general (for instance, implicit dependencies between architectural elements and features), bound to the domain engineering (stakeholder concept overlap, for example), the application engineering or the evolution of the variability.

Furthermore, (Pohl et al., 2006) identified 4 questions the documentation of the variability should answer:

- **What does vary?** Answering this question leads to variation points. For instance, the type of mechanism used to authenticate employees;
- **How does it vary?** Identifies the different instances of a variation point, i.e. the variants. For instance, the company can use *facial recognition*, *RFID* tags and so on;
- **Why does it vary?** Highlights the motivation behind a variation point or variant. For example, some privacy laws might discourage the use of facial recognition;
- **Who is it documented for?** What is the target group of a variation point/variant. For instance, the customer.

As part of *variability management*, *variability modelling* is critical to the success of a SPL. It regroups techniques to represent the product line variability in order to provide a better overview and understanding and to enable tools to take over some of the variability management tasks (Classen et al., 2011).

Due to its significant importance, many approaches have been developed to assist engineers in modelling the variability. These approaches vary in the way they represent the variability of the Software Product Line. *Orthogonal variability model* (OVM) for instance, relies on the notions of variation points and variants. The former represents an abstract property of the considered system (e.g. *Door lock*) while the latter is a specialization of the variation points (e.g. *Keypad* and *Fingerprint Scanner*). Text-based approaches are another example of representation.

Feature modelling however, as noted by Classen *et al.* , is the standard to variability modelling in Software Product Line Engineering (Classen et al., 2011). It is the most popular formalism to both model and reason about the variability of a system (Acher, Baudry, et al., 2013). By describing features at various levels of abstraction it can be used "to manage variability of artifacts that are produced in different development phases" (Acher et al., 2011).

This formalism, introduced by (Kang et al., 1990), represents "the standard features of a family of systems in the domain and relationships between them". It thus represents a set valid configurations (i.e, a set of sets of features) (Acher, Baudry, et al., 2013). To do so, it proposes to use a directed acyclic graph, usually a tree, where nodes are features and edges are top-down hierarchical decompositions of features (Classen et al., 2011).

The formalism, as introduced by Kang *et al.* , allowed the 3 different notations (Czarnecki et al., 2004) :

- *mandatory features*, indicating that all instances of the SPL **must** implement this feature;
- *optional features*, indicating that an instance of the SPL **can** implement this feature (or can choose not to do so);

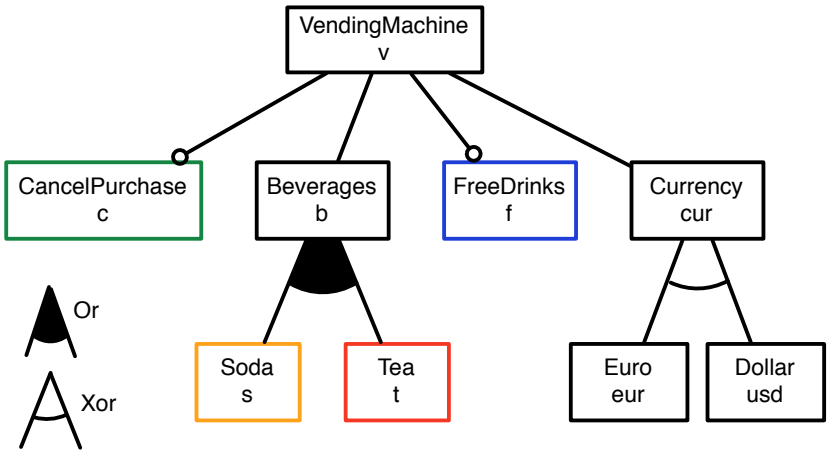

Figure 1.2.: Vending machine feature diagram, retrieved from (Classen et al., 2011)

- *alternate sub-features*, indicating that the sub-features **cannot** be simultaneously enabled. They can also be seen as a specialization of the super-feature (Kang et al., 1990): for instance, the alternate sub-features *Manual* and *Automatic* can be seen as specialization of the super-feature *Transmission*.

It was nonetheless extended later on to add, for example, *inclusive-or groups* allowing any combination of sub-features (with regards to the sub-features nature – mandatory or optional) (Czarnecki, 1998).

To illustrate feature models concepts, we use the *vending machine* case presented in Section 1.1. The feature diagram is presented in Figure 1.2.

First, this model proposes both optional (*CancelPurchase* and *FreeDrinks*) and mandatory (*Beverages* and *Currency*) features. That is, all instances of a *VendingMachine* offer beverages and rely on a currency but might (or might not) allow the cancelling of a purchase or allow free drinks.

Second, the feature model supports OR-groups and XOR-groups. For instance, a vending machine might offer Soda only, Tea only or both Soda and Tea. However, it can only accept Euro or Dollar but not both simultaneously.

In order to support engineers in modelling the variability, many tools have been introduced. These tools are either presented as Eclipse plug-ins (such as *FeatureIDE*[3], *fmp*[4] or *Pure::variants*[5] for instance) or standalone projects (e.g. FAMILIAR (Acher, Collet, et al., 2013)). These tools offer, beside the possibility to graphically model the variability, built-in reasoners to compute different statistics such as the number of valid configurations, for example.

[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[4] http://gsd.uwaterloo.ca/fmp
[5] https://www.pure-systems.com/products/pure-variants-9.html

We now discuss some forms of variability implementations at the code level.

1.2.2. Variability implementation

As we previously discussed, the variability can be bound at different time: compilation, link, runtime or update (or post-runtime). Moreover, the implementation of the variability can take many shapes and forms.

As presented in (Gacek & Anastasopoules, 2001) some techniques are more suitable for compile or link time while others are difficult or plainly impossible to set up at other binding times. Pohl *et al.* also argue that the choice of binding time and the modelling of the variability are independent, and that it is rather "a consequence of decisions made during design and realisation" (Pohl et al., 2005). We briefly summarize some variability implementation techniques here-after:

- **Aggregation/Delegation**: object-oriented technique in which an object may reference other objects (its *parents*) and if it receives a message for which it has no matching method then it forwards the message to its parents (Kniesel, 1999);
- **Inheritance**: concept of object-oriented programming where a class (*child*) can *extend* another one (*parent*) to benefit from the same fields and methods. A functionality assigned to the parent superclass can then be extended (modified) by the subclass child;
- **Parametrization**: technique that aims at maximizing program reuse by storing programs in a most general possible form (Goguen, 1984). The behaviour is then bound to the value of the parameters;
- **Conditional compilation**: mechanism enabling (or disabling) the compilation of code segments (Pohl et al., 2005);
- **Aspect-oriented programming**: technique aiming at regrouping cross-cutting functionalities (*aspects*) in consistent entities, weaved together into a single piece of code prior to the actual compilation (Pohl et al., 2005);
- **Makefiles**: link-time implementation mechanism relying on the use of *makefiles*, executables files allowing different sets of compilations/linkages based on given parameters (thus enabling the realisation of variation points) (Pohl et al., 2005).

1.3. Variability-intensive systems

Software Product Lines have disseminated variability-management ideas everywhere leading to the dawning of *variability-intensive systems* or *highly-configurable systems*.

Linux, with more than 14,000 features (Gazzillo, 2015), is a classic large-scale example As are Apache web server and the GNU compiler collection. These 3 common cases all rely on the C language.

However, C is not the only option for configurable systems. Plug-in based systems, for example, offer other examples of variability-intensive systems. For instance, Drupal[6], a content management system developed in PHP, supports 96,768 different configurations (Sánchez et al., 2013a). Another well known case is Wordpress[7], another PHP-based blog software offering 25,000 plugins (Nguyen et al., 2014a).

With this kind of systems, the variability is introduced at different levels and not just at the functional one. The variability can impact non-functional properties of the products for instance (Beuche et al., 2004).

[6] `https://www.drupal.org/`
[7] `https://wordpress.org/`

2. Software Product Line testing

As shown by several systematic studies – for instance (da Mota Silveira Neto et al., 2011), (Engström & Runeson, 2011) and (Lamancha et al., 2013) – a lot of effort has been put on SPL testing. We present in this Section the state of the art of known Software Product Line testing techniques.

First, we explore two vertices of the *Product Line Analysis cube* (Von Rhein et al., 2013). This model classifies analyses along three axes: *single product*, *product-based* and *family-based* approaches. We focus on the latter two, respectively presented in Section 2.1 and in Section 2.2. Moreover, regarding the product-based vertex, we explore techniques that tend to reduce the cost of testing in the presence of variability, and particularly sampling techniques.

Then, we also present in Section 2.3 another key challenge of SPLs: the *evolution of the product line*. This evolution can occur at different level (code, mapping, variability model and so on) and thus requires a special attention.

2.1. Products' analyses

Product-based analysis aims at generating and analysing each product individually, using a standard analysis method (Thüm et al., 2012). As noted by (Von Rhein et al., 2013), product-based analysis strategy focus on each product individually without taking into account the variability; it has been resolved using sampling techniques or by enumerating all products. One of the advantages of this method is that single-product analysis tools can be reused.

One approach to product-based analysis is the use of formal methods. The idea is to prove correctness properties in the specification at the product line level such that all derived products satisfy the same properties, without needing to enumerate all of them (ter Beek et al., 2015; Classen et al., 2013).

Another approach is to brute-force the generation of all products. However, product lines usually allow large number of products and the brute-force approach is only conceivable for smaller size SPLs. A more reasonable approach is to select and sort the fittest set of products to test according to given criteria in order to detect as much bugs as possible.

For instance, as mentioned in (Sánchez et al., 2015) Debian Wheezy[1], a Linux distribution,

[1] `http://www.debian.org/releases/wheezy/`

provides 37,000 packages and constraints leading to billions of potential configurations. In this context, the derivation of all products is inconceivable.

To address this problem, researchers have proposed various techniques to reduce the cost of testing in the presence of variability.

We present in next sub-sections some of known techniques that address this challenge. First, we present and compare *sampling techniques* that reduce the number of configurations to test. Second, we present test case techniques that aim at reducing the test space and at increasing their effectiveness. (Henard et al., 2014; Guo et al., 2011; Yoo & Harman, 2007; Sanchez, 2012). Third, we show that *unit testing* is the prevalent technique used in the testing culture. Fourth, we present the automated generation of test cases in product-level functional testing. Finally, we introduce quality attributes prediction.

2.1.1. Configurations selection

As previously mentioned, the derivation of all products is not feasible for most industrial size SPLs. To this end, many research effort has been put on selecting the right subset of configurations to test. These configurations selection techniques are called *sampling techniques*.

Features selection technique

As highlighted by (Deelstra et al., 2004), the derivation of individual products is costly in time and effort. Engineers commonly use a feature model to capture and document the commonalities and variabilities of the underlying software system in SPLs. As explained in (Guo et al., 2011), a key challenge when using a feature model to derive a new SPL configuration is to determine how to find an optimized feature selection that minimizes or maximizes an objective function, such as total cost, subject to resource constraints.

This selection technique, as explained in (Jain & Zongker, 1997), can reduce the cost by reducing the number of features and can also provide a better classification accuracy due to finite sample size effects but the need of a true optima subset is essential. The key is to find a performing feature selection.

Sampling techniques

To reduce the number of products to test, one popular research direction is to use Combinatorial Interaction Testing (CIT) techniques (Cohen et al., 2008; Lopez-Herrejon et al., 2015) and pairwise (generalized to t-wise) criteria (Lochau, Oster, et al., 2012; Lopez-Herrejon et al., 2013; Marijan et al., 2013; Pérez Lamancha & Polo Usaola, 2010; Perrouin et al., 2011). Over the years, several tools have been developed and support

pairwise based selection on the feature model (Hervieu et al., 2011; Johansen, Haugen, & Fleurey, 2012; Johansen, 2016).

In order to support larger t values, as well as larger feature models, other search-based heuristics have been proposed (Al-Hajjaji et al., 2016; Henard et al., 2014; Ensan et al., 2012; Sayyad et al., 2013; Sanchez et al., 2014; Parejo et al., 2016). All of those CIT, t-wise, and other search-based techniques make the hypothesis that faults come from interactions between few features and try to select an adequate set of products to test in order to cover as much feature combinations as possible. They have been extensively validated on a large number of feature models, with different sizes, and coming from different sources. However, very few evaluation have actually included building the set of products to test in their process.

Comparison of sampling strategies

The difficulty to assess all the possible configurations in the general case lead to the development of various sampling techniques, differing by their coverage criteria. E.g., pair-wise (Yilmaz et al., 2006; Cohen et al., 2008; Perrouin et al., 2011) or dissimilarity amongst configurations (Henard et al., 2014)), use of exact algorithms (Johansen, 2016; Hervieu et al., 2011), or metaheuristics (Garvin et al., 2011; Sayyad et al., 2013; Parejo et al., 2016; Henard et al., 2014).

Perrouin *et al.* compared two exact approaches on five feature models of the SPLOT repository w.r.t to performance of t-wise generation and configuration diversity (Perrouin et al., 2011). (Hervieu et al., 2011) also used models from the SPLOT repository to produce a small number of configurations. Empirical investigations were pursued on larger models (1,000 features and above) notably on OS kernels (e.g., (Henard et al., 2014; Johansen, Haugen, & Fleurey, 2012)) demonstrating the relevance of metaheuristics for large sampling tasks (Le Traon, 2015; Ochoa et al., 2017). However, these comparisons were performed at the model level (there was no product associated to these models) using artificial faults.

Several authors considered sampling on actual systems, thus dealing with real faults. Johansen *et al.* extended his SPLCAT tool with weights and to test the Eclipse IDE and an industrial and reverse vending machine system (Johansen, Haugen, Fleurey, Eldegard, & Syversen, 2012). Steffens *et al.* applied the Moso-Polite pairwise tool ((Oster et al., 2011a)) on an electronic module allowing 432 configurations to derive metrics regarding the test reduction effort. Additionally, they also exhibited a few cases where an higher interaction strength was required (3-wise). Sanchez *et al.* modelled a subset of the Drupal framework and (manually) examined how faults (extracted from the Drupal's issue tracking system) was related to feature interactions: amongst 390 faults, 11 were related to 2-wise interactions, 1 to 3-wise interactions, and the rest was related to single features (Sánchez et al., 2013b). A further analysis on this case revealed that with respect to 3392 faults, 160 were related to up to 4-wise interactions (Sánchez et al., 2017). Medeiros *et*

al. compared 10 sampling algorithms on a corpus of existing configurations faults taken from a large set of configurable systems (Medeiros et al., 2016).

The authors found that 2-wise interactions covered a large subset of faults (as it is commonly assumed) but also made the case for higher-interaction strengths (seven in their study) and that sampling algorithms such as most-enabled-disabled are most efficient in that case. Our study adds more empirical evidence on the fact that higher-interaction is desirable to find all bugs. However, in our case the most-enabled-disabled strategy did select a high number of sample which greatly reduced its efficiency.

Despite the number of empirical investigations (e.g., (Ganesan et al., 2007; Qu et al., 2008)) and surveys (e.g., (Engström & Runeson, 2011; Thüm et al., 2014; da Mota Silveira Neto et al., 2011)) to compare such approaches, many focused on subsets to make the analyses traceable. Being able to execute all configurations led us to consider actual failures and collect a ground truth. It helps to gather insights for better understanding the interactions in large configuration spaces (Meinicke et al., 2016; Yilmaz et al., 2006).

2.1.2. Test case

As defined by (Parejo et al., 2016), "Test case is defined as a configuration of the HCS under test (i.e. a set of features) and a test suite is a set of test cases." In a context of most industrial size SPLs, test case technique faces the same problem, it is impossible to make test cases for all configuration. Indeed, test suites tend to grow in size as software evolve, often making it too costly to execute them entirely. To this end, and as presented in (Yoo & Harman, 2007), different approaches have been studied to maximise the value of the accrued test suite: *minimisation*, *selection* and *prioritisation*.

Test case minimisation technique

The goal of *test suite minimisation* technique (or *test suite reduction*) is to reduce the size of a test suite by eliminating redundant test cases from the test suite. It is a reduction technique that is used to create a temporary subset of a test suite, which can be used to permanently eliminate test cases. (Yoo & Harman, 2007; Rothermel et al., 2002). Some empirical studies (Rothermel et al., 1998, 2002; Wong et al., 1998) investigate the effect of test suite minimisation on the fault-detection capability of test suites.

Test case selection technique

Test case selection aims at reducing the test space by selecting an effective and manageable subset of configurations to be tested (according to some coverage criteria) and also seeks to reduce the size of a test suite as the *test case minimisation technique*. This technique is essentially similar to the *test suite minimisation* problem; both problems are about choosing a subset of test cases from the test suite. With this technique, the selection is

not only temporary (i.e. specific to the current version of the program), but also focused on the identification of the modified parts of the program (e.g new releases). In this case, test cases are selected. Indeed, it is more relevant for changed parts (which typically involves a white-box static analysis of the program code).(Yoo & Harman, 2007) *Test suite minimisation* is often based on metrics such as coverage measured from a single version of the program under test.

Test case prioritization technique

Finally, as explained in (Yoo & Harman, 2007), "*test case prioritisation* concerns ordering test cases for early maximisation of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases."

The big difference with the other techniques is it doesn't involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process in an order that attempts to increase their effectiveness at meeting some performance goal, e.g. accelerate the detection of faults.(Sánchez et al., 2015)

Test case prioritisation seeks to "find the ideal ordering of test cases for testing, so that the tester obtains maximum benefit, even if the testing is prematurely halted at some arbitrary point" (e.g. computational cost).(Yoo & Harman, 2007)

Different criterion can be used for the *test case prioritisation technique*. For instance, *structural coverage* (Rothermel et al., 1998) is a metric with the intuition that early maximisation of structural coverage will also increase the chance of early maximisation of fault detection. Or another example, *interaction testing* metric is used when there are multiple combinations of different components. A common example would be configuration testing, which is required to ensure that different combinations of environment, such as different operating systems or hardware options are correctly executed.(Yoo & Harman, 2007)

2.1.3. Testing culture in configurable systems

(Greiler et al., 2012b) investigated how developers test their plug-ins in the Eclipse framework. The prevalent technique used was unit testing of individual plug-ins without any systematic approach to handle combination with other plugins. Interviewed people complained about the difficulty to set up a proper integration environment, long execution times (few minutes), and lack of best practices.

2.1.4. Product-level functional testing

Product-level functional testing focuses on functional properties of each product individually and relies on existing single-product analysis tools to perform analysis.

Due to the usually huge numbers of products in SPLs, researchers have proposed to derive test cases from product line scenarios and use cases (e.g., (Nebut et al., 2006; Oster et al., 2011b)) promoting the reuse of test models, adapted to the SPL context, with the goal to reduce testing costs.

For instance, (Nebut et al., 2006) present an approach to automate the generation of application system tests, for any chosen product, from the system requirements (UML use cases) of a SPL. Their idea behind this approach is to describe functional variation points at requirement level to automatically generate the behaviours specific to any chosen product and at the end, provide the automation of generated system test cases.

2.1.5. Feature-related quality attributes

Recent research shifts from functional properties to non-functional characteristics of products. The goal is to predict performance – such as response time, throughput, resources consumption and so on – of a given product. (Sarkar et al., 2015; Siegmund et al., 2012, 2013, 2015). In these researches, as for functional testing, they also face combinatorial explosion issues.

These analysis methods rely on statistical learning (Guo et al., 2013) and regression methods (Valov et al., 2015), or mathematical models to predict and detect (undesired) performance-relevant feature interactions (Zhang et al., 2016).

2.2. Family-based analyses

Model-based testing approaches use behavioural models of the product line to generate test cases for the different products: (resp.) delta-oriented product line testing (Lochau, Schaefer, et al., 2012; Lachmann et al., 2015) and featured transition system based (Devroey et al., 2014, 2015, 2016) approaches use (resp.) state machines and transition systems in order to capture the common and product specific behaviour of the product line. At the code level, variability-aware parsers (Kästner et al., 2011), variational structures (Walkingshaw et al., 2014) and type-checking (Kastner & Apel, 2008) are of interest. They indeed enable *variability-aware testing* (Kästner et al., 2012) to, for example, evaluate a test case against myriads of configurations in one run (Nguyen et al., 2014a).

2.3. Product Line evolution techniques

From the evolution perspective, product lines represent an interesting challenge. Product lines developers, architects and engineers have to manage updates at different levels: the evolution of the variability model and the mapping to other artefacts (Dintzner et al., 2016); the evolution of the artefacts themselves which will impact several products (Neves et al., 2015; Sampaio et al., 2016); and the evolution of the configurator and configuration workflow.

In order to understand how existing SPLs are updated, Passos *et al.* recently studied the Linux kernel variability models and other artefact types co-evolution. They inspected over 500 Linux kernel commits spanning almost four years of development and collected a catalogue of evolution patterns, capturing the co-evolution of the Linux kernel variability model, Makefiles, and C source code. They extracted general findings (e.g. most features of Linux kernel are modular and cause little scattering when added, variability evolution of the Linux kernel follow systematic patterns, etc) to guide further researches and tool development.

Part III.

Case study: JHipster

3. The case of JHipster

To support this study and address the research questions presented in Chapter 4 we introduced a new case study for the research community: *JHipster*. This project relying on different complex technologies, we present in Appendix B a short description of each framework.

3.1. Project background

JHipster is an open source generator for web applications. Under Apache 2.0 license, the source code is available at: `https://github.com/jhipster/generator-jhipster`. Initiated in 2013 by Julien Dubois, it was publicly released in December of that year. JHipster's objective is to assist the user in all phases of web applications development: the choice of the technologies to use, their integration in a complete and automated building process and the management of dependencies across the offered technologies (JHipster, 2016). It is used worldwide by large companies (such as Adobe or Google, for instance) as well as independent developers.

3.2. Core tasks and objectives

JHipster's web application creation process is divided in two phases: the *configuration process* and the *generation process* (Halin et al., 2017). These two processes can be used separately, as discussed later on.

3.2.1. Configuration process

The *configuration* of the web application is achieved via a command-line interface, presented in Figure 3.1. Through this interface, the user is prompted a list of questions to select the technologies that will be included in the configuration. The prompting of the different questions is handled by JavaScript files: *prompts.js*. JHipster comprises several of these JavaScript files and each of them handles a specific part of the configuration.

Figure 3.1.: JHipster command line interface

For instance, *client/prompts.js* allows the user to include *LibSass*[1] in its web application while *server/prompts.js* handles the database selection.

Although it doesn't provide a graphical user interface (*GUI*) like most configurators, in particular web-based configurators such as car configurators available on most constructors' websites, JHipster share many common characteristics with typical configurators.

Like all configurators (Boucher et al., 2013), JHipster assists the user in his decision making process by enforcing constraints, preventing conflictual decisions and propagating the decisions. It also relies on the same principles as its web-based counterparts (Boucher et al., 2013):

- It follows a *multi-step process*: options are presented in containers displayed one at a time; each step having to be validated before the next one becomes available;
- It comprises *group constraints*: the group defines the option(s) that can be selected. The *group constraints* can be exclusive (select only one type of database) or inclusive (*which one(s) of these testing frameworks do you want to use? if any*);
- It relies on *cross-cutting constraints*: the choice(s) made previously can impact the remainder of the configuration (e.g, if we configure a *monolithic* application then a database must be chosen, which is not mandatory with *microservices*);
- Like a minority of the cases studied by *Boucher et al.* , JHipster doesn't support *backward navigation*: once an option selected it cannot be changed;

[1] A CSS stylesheet preprocessor, more information can be found on the official website: `http://sass-lang.com/libsass`

The list of all questions and their different answers is presented in Table A.1 of Appendix A. As previously mentioned, a description of the different frameworks and technologies can be found in Appendix B.

The *configuration* process yields a JSON file (see Listing 3.1) used in the *generation* process. When calling JHipster (*yo jhipster*), Yeoman[2] first check if a *yo-rc.json* file is present. If so, this configuration process is skipped and Yeoman immediately launches the *generation* process. If a file is present but erroneous then parts of the configuration can be replayed. For instance, if in a JSON file the field *"buildTool"* is present but the value is neither *"maven"* nor *"gradle"*, then the question *"Would you like to use Maven or Gradle for building the backend?"* will be prompted.

Listing 3.1: `.yo-rc.json` example

```
{
  "generator-jhipster": {
    "jhipsterVersion": "3.6.1",
    "baseName": "jhipster",
    "packageName": "io.variability.jhipster",
    "packageFolder": "io/variability/jhipster",
    "serverPort": "8080",
    "authenticationType": "session",
    "hibernateCache": "ehcache",
    "clusteredHttpSession": "no",
    "websocket": "no",
    "databaseType": "sql",
    "devDatabaseType": "h2Disk",
    "prodDatabaseType": "mariadb",
    "searchEngine": "no",
    "buildTool": "maven",
    "enableSocialSignIn": false,
    "rememberMeKey":
       "e248fc12e226fc8ffcd4b6170c9d48dbc3b9ed12",
    "useSass": false,
    "applicationType": "monolith",
    "testFrameworks": [
      "gatling",
      "cucumber",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": false
  }
}
```

[2]Yeoman is a generator ecosystem on which JHipster relies. More information can be found on the official website: `http://yeoman.io/`

3.2.2. Generation process

JHipster *generation* process follows Yeoman workflow which aims at improving both user's productivity and satisfaction (Osmani et al., 2012). This workflow comprises tools to scaffold a project (*yo*), build and test this project (*Gulp* and *Grunt*) and to manage dependencies (*npm* and *Bower*)(Osmani et al., 2012).

To generate the complete application, Yeoman first uses template files. These files, present in JHipster generator, are different sorts of artefacts (XML, Java, JavaScript, CSS, ...) holding both shared content (common dependencies, for instance) and specific one (e.g, Java methods related to a specific database). Yeoman template files use *conditional compilation* (Gacek & Anastasopoules, 2001) to include or exclude parts of the code depending on the user's choices. This technique allows JHipster to use the same template for all configurations: the variability being resolved in the *configuration* process (the result is stored in the *.yo-rc.json* file), at compile time all technologies are chosen and the template is thus adapted. This *conditional compiling* is illustrated, as example, in the *pom.xml* template depicted in Figure 3.2. It shows, from line 151 to 157, the dependency *metrics-ehcache* which is to be included *if and only if* the configuration uses *Ehcache* as *Hibernate second level cache*. The other dependencies are included in all configurations, regardless of their content.

The generator then runs *npm* to retrieve needed dependencies and store them in a local *node_modules* folder, thus concluding the *generation* process.

```
140
141     <dependencies>
142         <dependency>
143             <groupId>io.dropwizard.metrics</groupId>
144             <artifactId>metrics-core</artifactId>
145         </dependency>
146         <dependency>
147             <groupId>io.dropwizard.metrics</groupId>
148             <artifactId>metrics-annotation</artifactId>
149             <version>${dropwizard-metrics.version}</version>
150         </dependency>
151         <%_ if (hibernateCache === 'ehcache') { _%>
152         <dependency>
153             <groupId>io.dropwizard.metrics</groupId>
154             <artifactId>metrics-ehcache</artifactId>
155             <version>${dropwizard-metrics.version}</version>
156         </dependency>
157         <%_ } _%>
158         <dependency>
159             <groupId>io.dropwizard.metrics</groupId>
160             <artifactId>metrics-graphite</artifactId>
161         </dependency>
```

Figure 3.2.: *pom.xml* template file excerpt

3.3. JHipster sub generators

JHipster actually offers many different (sub)-generators. Regarding the classical configuration/generation process, JHipster relies on 2 sub-generators, *jhipster-client* and *jhipster-server*, handling respectively the generation of client and server standalone applications.

JHipster Client
Accessible via the command *yo jhipster:client* or *yo jhipster –skip-server*, this sub-generator prompts questions 2, 16 and 17 presented in Table A.1. It then generates the AngularJS front-end code.

JHipster Server
Accessible via the command *yo jhipster:server* or *yo jhipster –skip-client*, this sub-generator handles back-end choices. It prompts all questions presented in Table A.1 except for questions 1, 3, 6, 16 and 18. The result of this sub-generator is the Spring Boot back-end only of a Web application.

As previously mentioned, JHipster relies on these two sub-generators to generate the different applications. It also prompts questions not included in these two generators: the type of the application to scaffold, the selection of the testing frameworks and the port number and path to the UAA server (respectively in microservices applications and UAA authentication based applications). It also uses the *language sub-generator* to add language support to the application. JHipster also recognize different optional *command-line options* to be used with the different generators. These options allow for instance to skip parts of the process. More information can be found on the official website: `https://jhipster.github.io/creating-an-app/#3`.

Besides these sub-generators, JHipster also offers an **Entity** sub-generator to generate entities and all related artefacts, such as Spring Service Beans. It also has its own Domain Specific Language (DSL): *JHipster Domain Language*[3].

3.4. JHipster's evolution

Since its beginning in 2013, JHipster has constantly grown throughout more than 100 releases. It now has 5378 stars on GitHub and can rely on a strong community and 256 contributors[4].

This constant evolution allow JHipster to offer up-to-date technologies both by offering new options (for instance, the ability to generate the infrastructure using Docker, since

[3] `https://jhipster.github.io/jdl/`
[4] Statistics retrieved from `https://github.com/jhipster/generator-jhipster` and `https://www.npmjs.com/package/generator-jhipster`, on November 16, 2016.

version 3.0.0) and by following the evolution of others (for instance, using LibSass instead of Compass, since version 2.4.0).

We present a list of available frameworks and technologies by JHipster version on our Github repository: `https://github.com/axel-halin/Thesis-JHipster/blob/master/Modeling/Comparatif%20JHipster.xlsx`

3.5. Motivation

Several reasons motivated the use of JHipster as a case study.

For starters, the problems we aim to address with this thesis (see Section 4.1) are common to most configurable systems. However, and although variability is everywhere, there has always been a shortage of publicly available cases for assessing variability-aware tools and techniques.

This shortage can be explained by the industrial secrets, contained in historical Software Product Lines, that their owners do not want to disclose to a wide audience. The open source community has nonetheless contributed to large-scale cases such as Eclipse, Linux Kernels or web-based plug-in systems.

To assess accuracy of sampling and prediction approaches (bugs, performance), a case where all products can be enumerated is desirable. The aforementioned available case-studies, however, lack this property. Wordpress, for instance, offered 25,000 different plugins (in 2014) (Nguyen et al., 2014b) while Linux Kernel 2.6.28.6 consisted in 5426 features (She et al., 2010). JHipster presents itself as a more manageable case study with $\approx$ 163,000 configurations, about 50,000 more than Drupal (Sánchez et al., 2015).

Furthermore, as configuration issues do not lie within only one place but are scattered across technologies and assets, a case exposing such diversity is highly valuable. JHipster, as we present in Chapter 3, is diverse in term of technologies. More so than Linux, for instance, which relies mainly on C language and where part of the variability can be extracted with *KConfig*, a language used to model the variability and the dependencies among configuration options (She et al., 2010).

Although similar studies have been conducted in the web domain (for instance, (Sanchez et al., 2014)), JHipster offers interesting assets beyond replication studies: *(a)* it covers key aspects of product line development, variability, product derivation and evolution; *(b)* the number of configurations ($\approx$ 163,000) is large enough to require automated derivation support (on top of Yeoman) but small enough to be enumerated through distributed computing facilities yielding exact results to assess various kinds of analyses; *(c)* it allows to address variability modeling and configuration challenges across technological spaces (Jin et al., 2014).

Moreover, we believe JHipster is a good candidate, both to devise new techniques and to assess existing ones (Halin et al., 2017), especially in *product-based* and *family-based* analyses.

As noted in (Jin et al., 2014), configuration issues can happen everywhere. The variety of technologies at work in JHipster products offers an opportunity to study such aspects and to study different kinds of interaction bugs. As we will see in Chapter 4.3, the variability model integrates information from different files.

Then, as opposed to plug-in-based cases, where test cases are either optional or depending on the will of developers (Greiler et al., 2012a), each JHipster application comes with test cases. In particular, Cucumber[5] supports early testing in the form of scenarios. Integration with code coverage tools is also available.

JHipster is also interesting for quality analyses. Web-apps are particularly interesting cases for performance, since this quality attribute has a direct influence on Websites' successes. Performance is not the only quality attribute that can be studied: security is also key especially for e-commerce websites. We will not cover such quality analyses in this master thesis but the variety of analyses possible on the JHipster case was an enabler for the selection of this case.

Finally, with more than 140 releases since 2013, JHipster is under active development and evolution. Therefore a challenge for researchers is to devise automated means to update the JHipster feature model. As opposed to the Linux case, where part of the variability model can be extracted from KConfig, several JavaScript files are necessary to build it, pushing for more versatile variability inference techniques.

[5] `https://cucumber.io/`

4. Research method

4.1. Research questions

This master thesis aims at answering 3 main research questions. First, we want to *exhaustively* test a configurable system and see what lessons can be learned from such an experiment. Then, we want to *compare the results of different sampling techniques comparatively to a ground truth.* Finally, we will extract from this study the *most-cost effective sampling strategy* and *discuss some recommendations* for the developers of the case study.

Those questions, although centred around JHipster as a case study, are in fact common to most configurable systems.

As previously mentioned (see Section 2), a key challenge in configurable systems is to ensure that all products are correct (no defects). Due to the sheer number of available products, a generally accepted idea appeared in Software Product Line Engineering (SPLE): it is unnecessary – and often even impossible – to test each product individually and sampling methods yields sufficient representative subsets of products.

With JHipster, the testing of the complete configuration space is within reach of current distributed (Grid) architectures. An immediate research question is to characterize the *cost* of an exhaustive and automated testing strategy:

- **(RQ1.1)** What is the cost of engineering an infrastructure capable of automatically deriving and testing all configurations?
- **(RQ1.2)** What are the computational resources needed to test all configurations?

To address these questions we will attempt to generate and test all JHipster configurations in order to get a ground truth to compare different sampling strategies and to answer the general question: **Is it worth testing all configurations?**

A second line of research is to qualify and quantify the configuration defects:

- **(RQ2.1)** How many and which sorts of failures/faults can be found in all configurations?

By collecting a *ground truth* or reference of defects, we can measure the effectiveness of sampling techniques. For example, is a random selection of, e.g. 50, configurations as effective to find failures/faults than an exhaustive testing? We can address this research question:

- **(RQ2.2)** How sampling techniques compare to in terms of effectiveness?

Finally, we can put in perspective the typical *trade-off* between the ability to find configuration defects and the cost of testing and address our final research question:

- **(RQ3)** What is the most cost-effective sampling strategy?

By addressing this last research question, we can present our recommendations to the JHipster team regarding the testing of their generator.

4.2. Methodology

In order to address the different research questions, we developed an infrastructure capable of assessing each JHipster configuration. This development was split in several steps, and is summarized in Figure 4.1, which we will present in this chapter.

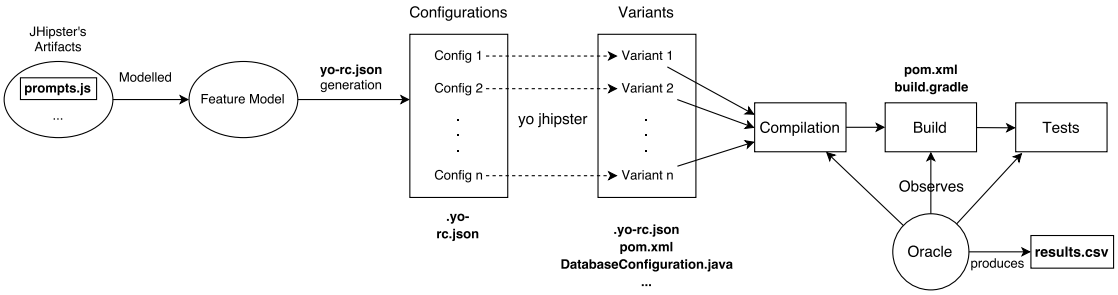

Figure 4.1.: Complete analysis workflow

The development was structured as follows: we first modelled the variability of JHipster in order to count the total of valid configurations (See Section 4.3), then we created an infrastructure that automate the generation and the tests of these valid configurations (See Section 4.4). Finally, due to the huge number of configurations, we had to scale up the distribution of the testing effort. For that, we used a cluster of machine (See Section 4.5).

We have developed this infrastructure capable of testing all configurations of JHipster at full-time during 4 months (from September 15th 2016 to January 15th 2017 at INRIA Rennes-Bretagne Atlantique, in the DiverSE team[1]). We used Github to track the evolution of the testing infrastructure. We also performed numerous physical or virtual meetings (with Slack) with our supervisors (Mathieu Acher, Gilles Perrouin and Xavier Devroey), which have supervised our efforts and provided us guidance based on their expertise in software testing and Software Product Line Engineering. Through frequent exchanges, we gathered several qualitative insights throughout the development. We

[1] `http://diverse.irisa.fr/`

also wrote in the early stages of our research internship with our supervisors an article *Yo variability! JHipster: a playground for web-apps analyses* that has been published and presented at the *11th International Workshop on Variability Modelling of Software-intensive Systems*. This article, available in Appendix H, presents our motivation to use JHipster in the research community and has been used as a starting point to write our master thesis. This paper presents also JHipster as a case for eduction which is not presented in this master thesis.

Besides, we decided not to report faults whenever we found them. Indeed, we wanted to observe whether and how fast the JHipster community would discover and correct these faults, without interfering on the JHipster project. We monitored JHipster mailing lists to validate our testing infrastructure and characterize the configuration failures in a qualitative way. We have only considered Github issues since most of the JHipster activity is there. Additionally, we used statistical tools to quantify the number of defects, as well as to assess sampling techniques. Finally, we crossed our results with insights from three JHipster's lead developers (a 1,5 hour semi-structured interview with the technical leader of JHipster by Skype and mail exchanges).

The constant evolution of JHipster, however, has lead us to restrain our study to a specific version of the configurator: its version **3.6.1**. Indeed, since the start of the study, and as of May 2017, JHipster was already in its 4.4 version. We selected the release 3.6.1 for 2 reasons:

1. at the start of the study (i.e, September 2016), it was among the latest available versions, thus offering the latest supported technologies;
2. by selecting a minor release we mitigated the risk of finding defects due to an early and unstable release

The release notes regarding this version can be found on JHipster official website: `https://jhipster.github.io/2016/08/18/jhipster-release-3.6.1.html`.

All resources related to this study can be found on-line: `https://github.com/axel-halin/Thesis-JHipster`. This repository regroups the source code we developed but also results files and other artefacts.

4.3. JHipster's variability modeling

To start the development of the testing infrastructure, we first had to capture JHipster's variability. To do so, we used the Feature Models formalism.

The selection of this formalism allowed us to rely on existing tools such as FeatureIDE[2] and FAMILIAR[3]. The former is an Eclipse plug-in supporting Feature-Oriented Software

[2] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
[3] `http://familiar-project.github.io/`

Development (FOSD) and specifically the design of Feature Models (Thüm et al., 2014). The latter is a Domain Specific Language (DSL) also allowing the design and reasoning on FMs (Acher, Collet, et al., 2013).

These tools offered, beyond the Graphical User Interface to draw the FM, built-in functions to assess the validity of the model or to count the number of valid products. Our internship mentor (Mathieu Acher, INRIA Rennes) being a part of the team which worked on FAMILIAR, we also had access to up-to-date source-code.

After a few experiments with the command-line configurator, we quickly noticed the scalability limitations of a brute force try of every possible combinations. Indeed, depending on the answer to a question, some following answers or questions can be deactivated.

As a result we considered the generator's source code, available on the official GitHub repository of the project[4].

JHipster being a Yeoman generator, it follows a specific structure: a *package.json* in the root folder and a *generators* folder where the main code is located. Among the artifacts of this folder are several *index.js* and *prompts.js* files. Each couple of these JavaScript files handles a specific part of the generator: *client* handles client side choices (such as the use of *LibSass*) while *server* handles the database selection, for instance. Furthermore, *index.js* handles the generation phase (the fetching of dependencies for instance) and *prompts.js* handles the configuration phase (i.e, the prompting of the questions). To assess and model JHipster variability, the latter proved more relevant.

After a thorough analyses of the JavaScript artefacts, we identified variability points (in the form of questions to be prompted during the configuration process) and constraints existing between them (conditions needed to be asserted before prompting a question).

Listing 4.1 is an excerpt of *server/prompts.js*. It illustrates the selection of the type of database. We can see that the user can select 3 types of database (SQL, MongoDB or Cassandra) or not to include a database in its configuration. Moreover, this question is only prompted if the application is a microservice (this is selected in a previous question).

There is also a similar question in the file to be prompted in other types of configuration (i.e, monolithic web-applications or microservice gateways).

Besides these JavaScript files, JHipster relies on conditional compilation to bind its variability. Conditional compilation is the main implementation mechanism for realizing variability. Developed as a Yeoman generator, JHipster relies on template files. These files (Java, HTML, XML, *etc.*) include conditions surrounding specific code snippets to be enabled depending on the considered configuration.

Listing 4.2 presents an excerpt of _DatabaseConfiguration.java template file. In the case of a MongoDB-based variant, this class will inherit from *AbstractMongoConfiguration*

[4] `https://github.com/jhipster/generator-jhipster/releases/tag/v3.6.1`

Listing 4.1: server/prompt.js excerpt

```
1  (...)
2  when: function (response) {
3      return applicationType === 'microservice';
4    },
5    type: 'list',
6    name: 'databaseType',
7    message: function (response) {
8        return getNumberedQuestion('Which *type* of database would you like to
             use?', applicationType === 'microservice');},
9    choices: [
10       {value: 'no', name: 'No database'},
11       {value: 'sql', name: 'SQL (H2, MySQL, MariaDB, PostgreSQL, Oracle)'},
12       {value: 'mongodb', name: 'MongoDB'},
13       {value: 'cassandra',name: 'Cassandra'}
14   ],
15   default: 1
16 (...)
```

(See line 9); in the case of SQL-based variants, on the other hand, some annotations (*EnableJpaRepositories* for example (See line 3)) will be activated (while the class won't inherit from *AbstractMongoConfiguration*, MongoDB and SQL being two exclusive features).

Listing 4.2: Variability in _DatabaseConfiguration.java

```
1  (...)
2  @Configuration<% if (databaseType == 'sql') { %>
3  @EnableJpaRepositories("<%=packageName%>.repository")
4  @EnableJpaAuditing(...)
5  @EnableTransactionManagement<% } %>
6  (...)
7  public class DatabaseConfiguration
8  <% if (databaseType == 'mongodb') { %>
9          extends AbstractMongoConfiguration
10 <% } %>{
11
12  <%_ if (devDatabaseType == 'h2Disk' || devDatabaseType == 'h2Memory') { _%>
13    /**
14    * Open the TCP port for the H2 database.
15    * @return the H2 database TCP server
16    * @throws SQLException if the server failed to start
17    */
18    @Bean(initMethod = "start", destroyMethod = "stop")
19    @Profile(Constants.SPRING_PROFILE_DEVELOPMENT)
20    public Server h2TCPServer() throws SQLException {
21       return Server.createTcpServer(...);
22    }
23   <%_ } _%>
24 (...)
```

From all these artefacts, we manually reverse engineered the feature model presented in Figure 4.2. We decided to model the variability as follows:

1. each multiple choices question is model as an *abstract feature*. These questions typically are "Which of these technologies do you wish to use?";
2. each answer is model as a *concrete feature*;
3. exclusive answers (select one and only one type of database, for instance) are mapped as alternate groups (testing frameworks selection is the only question allowing multiple answers, hence these features are represented by an OR-group);
4. yes or no questions are represented by optional features;

So, if we consider Listing 4.1 as an example we have: *database* as an optional abstract feature, with *SQL*, *Mongodb* and *Cassandra* as concrete alternate sub-features.

We also extracted several constraints between features, *when (...) return applicationType === 'microservice'* in Listing 4.1 is one of them. We synthesized a total of 15 constraints, presented in Figure 4.2.

Besides these 15 constraints we identified a total of 48 features (33 of them being concerned by constraints). We relied on FAMILIAR built-in functions to compute the total number of valid configurations: **162,508**.

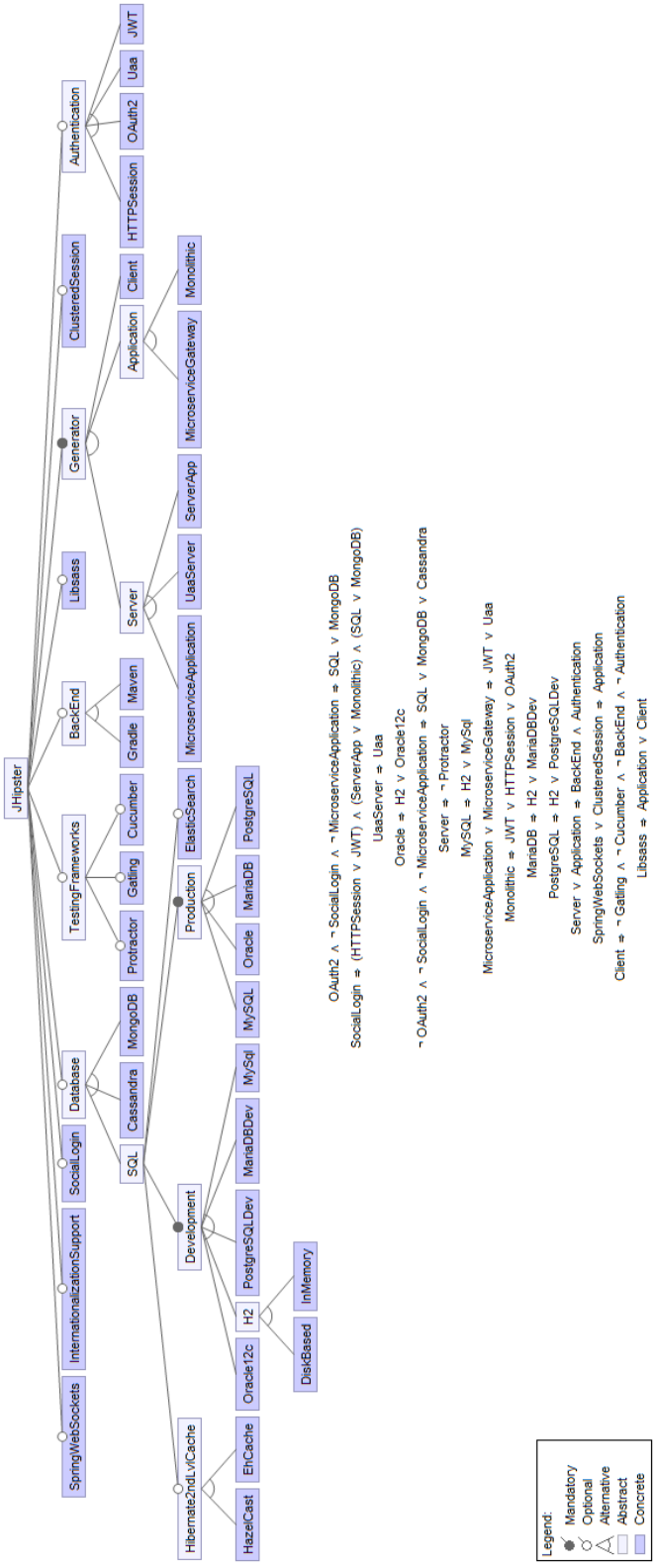

Figure 4.2.: JHipster 3.6.1 - feature model

4.4. Analysis workflow

With the variability model at hand, we started the development of a process to automatically generate and test all valid configurations.

During this development phase, we decided to refine the feature model presented in Chapter 4.3. This decision was motivated by technical and practical reasons.

First, we decided to exclude client and server standalones applications from the configuration space. This exclusion was motivated by a limited interest for users to consider the server (respectively client) without a client (respectively server). Moreover, failures most likely occur when both sides are inter-related and are then covered by monolithic web applications.

Second, we removed Oracle-based variants from the set of tested configurations. Oracle is a proprietary technology with technical specificities that are quite more complex to fully automate.

Third, whenever possible, we decided to include all testing frameworks (Cucumber, Gatling and Protractor) in each configuration. Due to some identified constraints, the inclusion of all three testing frameworks was only possible in monolithic web applications and microservice gateways; microservice applications only support Cucumber and Gatling. By including these testing frameworks, we prevented the testing of similar configurations: the inclusion of testing frameworks does not impact the configuration itself but merely adds support for additional tests (load balancing, etc.); if we did not set those features to mandatory, we would have had to typically handle 8 times the same configuration.

Finally, we added a new feature to the model. From JHipster 3.0.0 on, each web application can be deployed with the build tool (Maven or Gradle) or with Docker. However, it is not a feature like the others (Docker is not part of the configuration process) : each configuration includes Docker but we choose to deploy the web application with it or not.

In the rest of this thesis, we are considering the original feature model augmented with specialized constraints that negate red features (e.g., Oracle) and that includes green features (e.g., Docker). This update of the variability model reduced the number of configurations to assess to **26,256**. Figure 4.3 presents this updated variability model.

The development of the analysis workflow was achieved in 2 steps: the derivation, from the feature model, of the configurations and the testing of all these configurations.

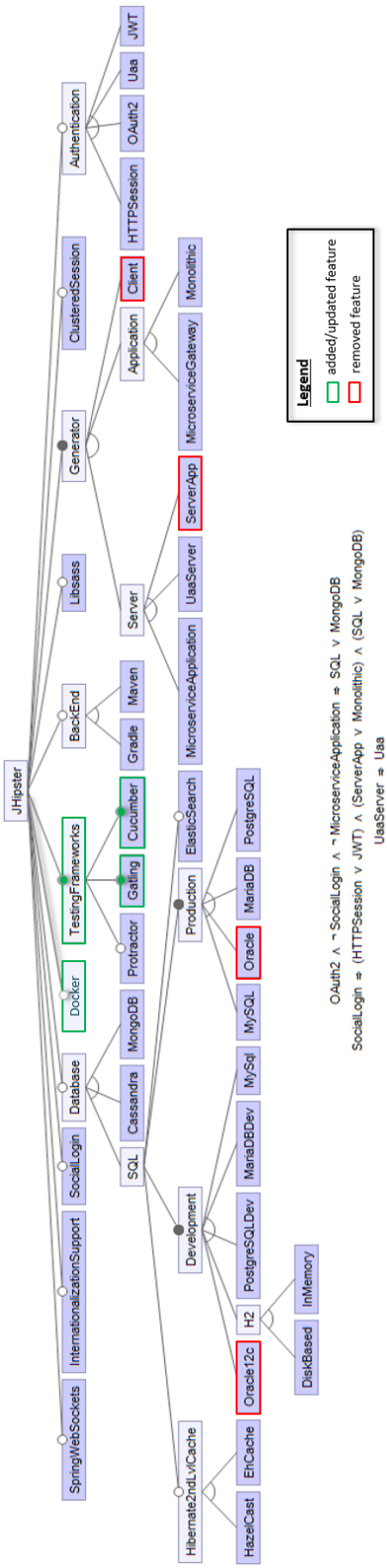

Figure 4.3.: JHipster 3.6.1 - updated feature model

4.4.1. Configurations derivation

This first engineering step consists in automatically deriving all valid (with regards to the Feature Model) configurations.

Concretely, we rely on FAMILIAR capabilities to enumerate all valid configurations. Our infrastructure then generate for each of them the matching *.yo-rc.json* file, each JSON file summarizing the configuration (as presented in Section 3.2.1). Based on the analysis of a few JSON files, we identified the specific fields the generator expects and were then able to generate the correct *.yo-rc.json* files.

This correctness is further asserted in the execution of the workflow. When calling the generator with a JSON file missing a key, the user will have to answer the relevant question. For instance, if we specify the key *DatabaseType* instead of *databaseType*, the generator will prompt us the database-related questions. The same will occur if we omit to specify a key.

Our infrastructure also generate at this step the different scripts used in the analysis workflow itself. Each script is linked to a step presented in Section 4.4.2. The scripts comprise common (*yo jhipster* for instance) and specific commands (*mvwn -Pprod* for example) depending on the configuration (specific commands are mostly dependent on the build tool, the testing frameworks, the use of Docker and on the type of database).

This step yields all JHipster directory (one per configuration) with all scripts necessary to execute the analysis workflow.

4.4.2. Variants testing

The second engineering step was the development of the analysis workflow itself which handles the testing of each variant. This workflow is summarized in Figure 4.4.

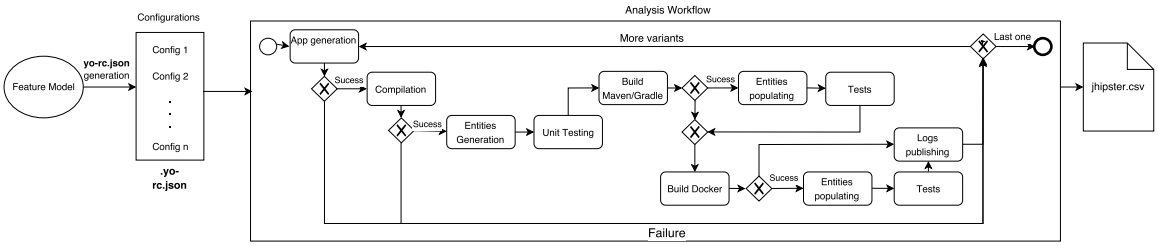

Figure 4.4.: Modelling of oracle tasks

In this workflow, each variant goes through several sequential steps. These steps are executed through the use of *bash* scripts and yield separate log files.

Besides these different steps, we developed 3 generic *oracles* to assess the result of the generation, the compilation and the build. These oracles seek stacktraces, error messages

or success messages in the log files. A failure to generate, compile or build leads to the skipping of some following steps (no need to compile the application if it fails to generate properly).

The workflow also comprises steps to improve the overall execution. For instance, before the generation of each configuration we paste a folder comprising all JavaScript dependencies. As presented in Section 3.2.2, JHipster relies on *npm* and *Bower* to retrieve dependencies. By including the *node_modules* folder in the working directory, this dependencies retrieving phase is skipped. This improvement of the workflow becomes interesting when scaling up the execution: we noticed a gain of several seconds to a few minutes (on a slow Internet connection).

The results and metrics (see here-after) of the different configurations are stored in a CSV-like file: a Google Sheet. This decision was motivated by limitations encountered while distributing the testing effort (see next Chapter). Before executing this workflow on a configuration, we assert it hasn't already been tested by checking the CSV content.

We present here-after the different steps of the analysis workflow.

Web app generation

The first step of the workflow is to **generate** the web application. This generation is based on the JSON file previously generated (Chapter 4.4.1) and consists in calling the generator as any user would: *yo jhipster*.

As we mentioned in Chapter 3.2, JHipster is both a configurator and a generator where the configuration yield a JSON file used to generate the matching application. By calling the generator in a directory containing a *.yo-rc.json* file, we can skip the configuration phase and directly scaffold the JHipster application.

This scaffolding both generates the artefacts based on the templates files and retrieve the needed dependency (already done with our improvement of the workflow).

This step produces a log file (*generate.log*) containing *Server app generated successfully* and/or *Client app generated successfully* if the generation succeeds. Our infrastructure checks the presence of error message of stacktraces in this log and report them in the main CSV output file if any occurs.

Our infrastructure also extracts from this step the time it took to be achieved. We found a mean time of *29.8* seconds (with generation times comprised between *2* and *475* seconds).

Web app compilation

With the application scaffolded we can start to assess its validity. To this end, our infrastructure starts by compiling it. JHipster supports two different build tools: *Maven*[5] and *Gradle*[6]. Depending on the configuration, our infrastructure then executes the matching command (*mvn compile* or *./gradlew compileJava*).

The log file of this step (*compile.log*) contains either *"BUILD FAILED"* or *"BUILD FAILURE"* if this compilation fails. If it does, our infrastructure stops the treatment of the current configuration (we can't build it if it cannot compile).

Our infrastructure also extracts error messages and stacktraces in case of failure. No matter the compilation result, our infrastructure extracts the time it took to be performed.

Moreover, prior to the compilation itself, our infrastructure runs database related scripts. These scripts start the database services needed in later steps (for instance, to deploy the web app), create tables and populate default entities (when need be). The different services need to be started when deploying the application with Maven/Gradle (see Section 4.4.2) and our infrastructure does so here to give it enough time to start.

Entity generation

After a successful compilation our infrastructure generates some entities to augment the web application (i.e to improve later test results).

The creation of entities in JHipster applications can be achieved with a sub-generator (see Section 3.3) that scaffold all necessary artifacts (including the creation of RESTful web services). We use this sub-generator to achieve this step (which is scripted too).

The sub-generator works like the main generator: our infrastructure uses a command-line interface to answer a few questions (name of the entity, attributes, relationships, etc.) and it yields a JDL file used to generate all artefacts. Similarly to the generation of the web application, our infrastructure calls the generator on a previously generated JDL file to skip the configuration process.

JHipster has its own Domain Specific Language called JDL (JHipster Domain Language), allowing the definition of entities, with an on-line editor[7]. The JHipster team provides an example (see Figure 4.5) which we used in this study.

This template, however, doesn't work for each database type. Indeed, NoSQL databases (MongoDB and Cassandra) cannot have relationships between entities. Moreover, Cassandra poses additional constraints. We thus use 3 different JDL scripts: one for MongoDB, one for Cassandra and one for SQL databases. This last one is in fact only used for

[5] `https://maven.apache.org/`
[6] `https://gradle.org/`
[7] `https://jhipster.github.io/jdl-studio/`

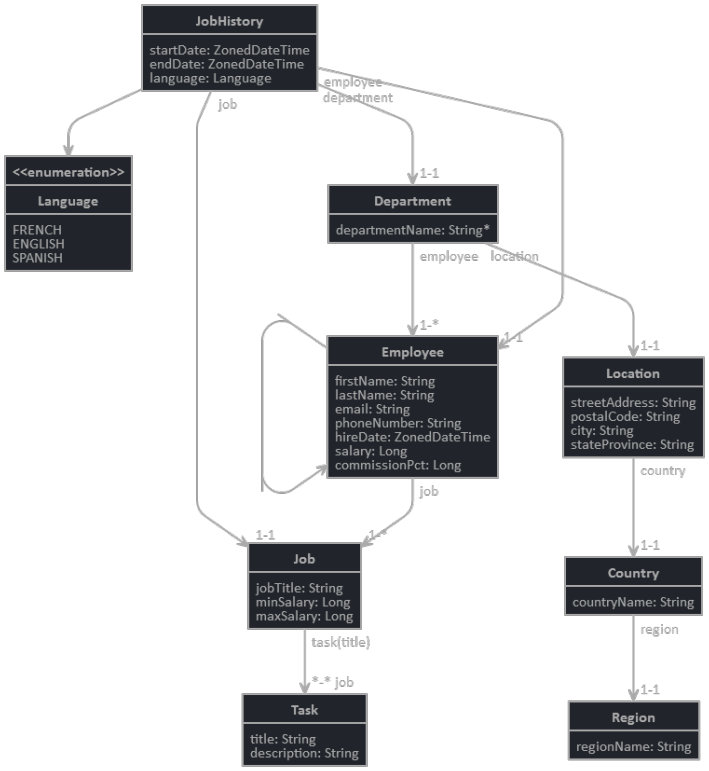

Figure 4.5.: JHipster JDL entities example (retrieved from the official website)

MySQL-based and PostgreSQL-based variants: MariaDB was not yet supported when we performed our experiment (it has been added since then).

We present in more details the constraints, the scripts and the models used for the different types of databases in Appendix C.

Common tests

Each web application, regardless of the configuration, comes with default tests which our infrastructure runs at this point.

First, each variant supports integration testing through *Spring Test Context* framework. These integration tests can be ran both with Maven (*./mvnw clean test* and Gradle (*./gradlew test*). The tests focus on common functionalities (accounts, users, audit API, etc.). From this execution, we extract the number of test failures, test errors and skipped tests.

Second, each web application support U.I testing with *Karma.js*[8]. These tests assert the

[8] `https://karma-runner.github.io/1.0/index.html`

correctness of the REST API by simulating the user interaction with the interface. It launches the server, runs typical user interface tests such as, for instance, log in attempts with correct and incorrect credentials and finally, kills the server. *Karma.js* is executed through *gulp.js*[9] (*gulp test*). Our infrastructure does not extract quantitative data from this test and we limit it to check the success (OK) or failure (KO) of KarmaJS tests.

From this first testing step, our infrastructure also extracts (whenever possible) the percentage of test coverage with *Jacoco*[10] and *lcov*[11].

Web app deployment

At this point, the web application is ready to be deployed. Two options are available to deploy the web app: using the build tool (*Maven* or *Gradle*) or using *Docker*[12]. We decided to use both options; each configuration is then built twice.

The main difference between building an application with Docker or with Maven/Gradle is that, while the former take care of starting additional required services (JHipster Registry, UAA server, etc.), the latter relies on the user to start those services. Both alternatives require the execution of specific scripts.

Docker:

- Prior to the build itself (*docker-compose*) our infrastructure stops all running database services. If the database service is running when attempting to deploy with Docker, an error will occur (*the port is already allocated*);
- After the build, our infrastructure removes all Docker images (*docker rmi $(docker images -q)*). This is particularly needed for database related images: 2 MySQL-based variants, for instance, don't share the same schemas depending on the configuration (the authentication mechanism for example)

Maven/Gradle:

- If the configuration we are trying to build is a microservice (gateway, Application or UAA server), our infrastructure first needs to launch the JHipster Registry[13];
- If a microservice uses UAA as authentication type, our infrastructure also needs to start the UAA server;
- Similarly to Docker build, our infrastructure removes all created databases.

[9] `http://gulpjs.com/`
[10] `http://www.eclemma.org/jacoco/`
[11] `http://ltp.sourceforge.net/coverage/lcov.php`
[12] `https://www.docker.com/`
[13] `https://github.com/jhipster/jhipster-registry`

The execution of the build being thread blocking (the command line used to run the build command will block until the process is killed or the build failed), our infrastructure starts another thread to assert the success or failure of this step. This thread checks the output log file every 2 seconds seeking either *Application 'jhipster' is running!* or any message indicating a build failure.

If the build succeeds our infrastructure executes the tests presented in Section 4.4.2 and then kill the server, if it fails we directly kill the server. This "kill" consists in terminating the process running on certain ports (JHipster application, JHipster Registry, etc.) to make sure there is no remnants of the execution After the completion of the build phase our infrastructure also drops the database so that it is re-created on the next configuration (depending on the configuration, some tables are not the same)

Our infrastructure extracts from this phase the results (OK/KO), the time it took the application to deploy (when building with Docker, our infrastructure takes into account the time needed to package the application as well as the time needed to run the container) and the stacktrace(s)/error message(s) when the build fails. When building with Docker, our infrastructure also extract the size of the Docker image.

Entity populating

When entities are defined in JHipster web application, the default front-end (common to all configurations) offers the possibility to populate the database.

To improve test results, we decided to use this possibility to add 10 elements per entities. Our infrastructure rely on Selenium[14] to simulate user interaction with the interface and add those entities.

Web app testing

The final phase of the analysis workflow consists in running additional tests on the application.

To this end, our infrastructure relies on the three different testing frameworks offered by JHipster. As we previously discussed, and whenever possible, we include all testing frameworks in the configurations allowing us to run additional tests.

Protractor[15]:
Similarly to KarmaJS tests, Protractor focus on UI testing. The tests simulate user's interaction with the default graphical user interface (GUI) generated by JHipster. They typically tests the log in, the access to the different pages when authenticated, etc.

[14] `http://www.seleniumhq.org/`
[15] `www.protractortest.org/`

Our infrastructure extracts from these tests the total number of tests ran and the number of failures.

Cucumber[16]:
Cucumber offers Behaviour-driven development (Solís & Wang, 2011). It consists in assessing the correctness of user defined scenarios (some are defined by the JHipster team). Listing 4.3 illustrate one of these scenarios.

When generating entities with JHipster sub-generator, some tests on the entities are automatically added. These tests are common CRUD (*c*reate, *r*ead, *u*pdate and *d*elete) operations: updateEntity, getEntity, getNonExistingEntity, deleteEntity and getAllEntity (where Entity is the name of the selected entity). Besides these user-defined entities related tests, JHipster provides common tests regarding the accounts and the users (trying to access a page without being logged in, testing the registration, and so on).

Listing 4.3: Cucumber scenario example

```
1 Feature: User management
2
3         Scenario: Retrieve administrator user
4                 When I search user 'admin'
5                 Then the user is found
6                 And his last name is 'Administrator'
```

Gatling[17]:
Gatling is a Scala-based stress testing tool (Šmeral, 2014) used to evaluate the performance of the application. It relies on scenarios and tests CRUD operations on the defined entities. Listing 4.4 presents an example of such scenario.

Gatling outputs metrics such as the number of requests, the mean response time and so on.

[16] https://cucumber.io/
[17] gatling.io/

Listing 4.4: Gatling scenario excerpt

```
1    val scn = scenario("Test the Country entity")
2        .exec(http("First unauthenticated request")
3        .get("/api/account")
4        .headers(headers_http)
5        .check(status.is(401))
6        .check(headerRegex("Set-Cookie", "CSRF-TOKEN=(.*);[P,p]ath=/")
7                .saveAs("csrf_token"))).exitHereIfFailed
8        .pause(10)
9        .exec(http("Authentication")
10       .post("/api/authentication")
11       .headers(headers_http_authenticated)
12       .formParam("j_username", "admin")
13       .formParam("j_password", "admin")
14       .formParam("remember-me", "true")
15       .formParam("submit", "Login")).exitHereIfFailed
16       .pause(1)
17       .exec(http("Authenticated request")
18       .get("/api/account")
19       .headers(headers_http_authenticated)
20       .check(status.is(200))
```

Logs publishing

Before moving on to the next configuration to process, our infrastructure regroups all the log files in a *tar* archive which our infrastructure then uploads to a Google Drive.

This allows us to have access to the complete logs[18] of every phase and ensure traceability.

4.5. Scalability

We initially started the execution of the analysis workflow locally, on a dedicated machine. Although it provided direct access to the process and its output, thus allowing us to monitor the behaviour of the workflow, it quickly became obvious this single machine wouldn't be sufficient to assess all configurations. Indeed, 2 to 3 days were required to assess about 300 configurations. So, with about 100 configurations per day, it would take *8 months, 2 weeks and 5 days* to test all *26,256* configurations on a single machine.

This observation, combined to the sheer number of configurations, led us to consider options to scale up the execution. We decided to rely on Grid'5000[19], a large-scale testbed offering a highly re-configurable, controllable and monitorable experimental platform (Balouek et al., 2012).

We present the resources the Grid'5000 supercomputing infrastructure offers in Appendix E.

[18] These files are available at: `https://drive.google.com/open?id=0B3EoDLh4drusa3hEaTlzbk5teE0`
[19] `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`

4.5.1. All-inclusive testing environment

The distribution of the testing effort strengthened the need of a common environment. Indeed, each JHipster configuration requires to use specific tools and pre-defined settings. Without them, the compilation build or execution cannot be performed.

With the distribution of the testing effort on several nodes in mind, we built an integrated environment capable of deriving any JHipster configuration.

In practice, this environment consisted of a Debian Jessie image. We installed on it each required database (MySQL, PostgreSQL, MariaDB, MongoDB and Cassandra) and other required tools (Maven, Gradle, Docker, etc.).

This configuration required a substantial engineering effort and was based on numerous tries and errors: we executed the workflow with several configurations to assert the correct configuration of the environment. If the tested variant did fail we then sought the cause and if need be adapted the Debian image.

At the end we converged on an all-inclusive environment.

4.5.2. Distributing the execution

With this common system image, we started to experiment with Grid'5000.

Grid'5000 offers clusters of machines (or nodes) scattered across France (see Figure 4.6). It relies on a system of reservation where we specify the number of nodes and the period of time for which we wish to use them (in Rennes alone, 173 nodes are available for reservation).

Before starting to use the reserved nodes, an environment image must be deployed. As we mentioned, we chose a Debian Jessie image. Once this deployment achieved, we can freely access each machine through *ssh* and start the processing of JHipster variants.

When the timer expires, each reserved node is reset and unreachable. Every data stored locally on the machine is then lost (this specificity motivated the use of Google API to store the results).

Although Grid'5000 allowed us to split the testing effort (thus reducing the overall testing time) it also suffers drawbacks.

The first limitation is due to its usage policy[20]. The reservation of machines, for large scale experiments (i.e requiring several machines for long period of time), is only possible at night (between 7 PM and 9 AM, to be exact) or during the weekend. This requirement leads to high reservation rates for weekends and some competition to book machines in these periods.

[20] `https://www.grid5000.fr/mediawiki/index.php/Grid5000:UsagePolicy`

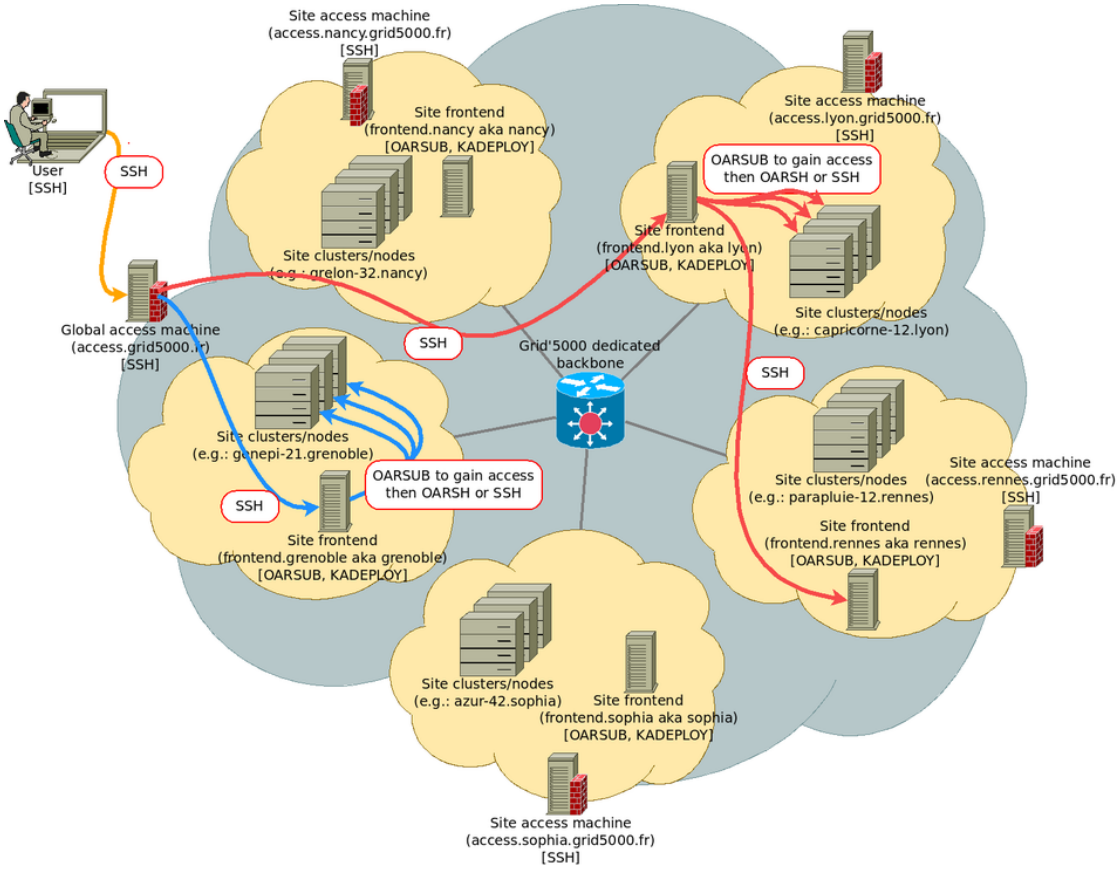

Figure 4.6.: Grid'5000 overview (retrieved from the official website)

Secondly, Grid'5000 also poses network limitations. We initially considered using a Client/Server architecture to send the results from the nodes on the Grid to a dedicated machine at INRIA Rennes. This solution would allow us to retrieve all logs and results from the workflow. However, we couldn't implement this architecture: Grid'5000 nodes were unable to reach INRIA network. To avoid losing too much time and effort on finding a workaround to this limitation, we decided to rely on Google API as presented in Section 4.4.2 to record our results in Google Spreadsheets.

A third limitation we faced was the low available disk space on the reserved nodes. Indeed, the disk space is split in several partitions and depending on the directory in which we are, we switch through these different partitions (see example in Listing 4.5). We can see that the most a partition offers is less than 400 Gb. Although this free space is quite reasonable, it quickly become insufficient depending on the number of variants tested (each variant weight about 400 Mb). Moreover, we found that variants using ElasticSearch wrote data in a folder belonging to */dev/sda3* partition which only offers 11Gb of free disk space.

Listing 4.5: *parapide-2* partitions

```
root@parapide-2:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        22G   11G   11G  51% /
udev             10M     0   10M   0% /dev
tmpfs           4.8G  8.8M  4.8G   1% /run
tmpfs            12G  4.0K   12G   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs            12G     0   12G   0% /sys/fs/cgroup
/dev/sda5       418G  1.8G  395G   1% /tmp
```

Finally, the Grid'5000 being a network of machines it occasionally suffers from unavailability due to maintenance, slow upload/download rate, and so on.

Part IV.

Results

5. The cost of testing all configurations

We address in this Chapter the first research question (see Section 4.1): What is the cost of testing all configurations of JHipster?

To this end, we first describe the cost of our engineering efforts in building a fully automated testing infrastructure for all JHipster variants (**RQ1.1**).

We then evaluate the computational cost of such an exhaustive testing (**RQ1.2**) by describing the necessary resources (man-power, time, machines) and reporting on encountered difficulties as well as lessons learned.

5.1. Engineering effort

The development of a complete derivation and testing infrastructure required a substantial engineering effort.

As described from Chapter 4.3 to Chapter 4.5, this effort was split across several steps. We quantify for each of them the required effort.

The first step was the modelling of JHipster variability. The elaboration of the first major version of the feature model took us about **2 man-week** based on the analysis of the JHipster code and configurator.

Then, based on this variability model, we initiated the development of the testing workflow. We added features and testing procedures in an incremental way. The effort spanned on a period of **2 man-month**.

In parallel of this development we built the Debian image. It also lasted a period of **2 man-month** for identifying all possible tools and settings needed. It should be noted that we performed medium-scale experiments on a local machine for prototyping and validating our solution.

With the workflow fully implemented, we decided to deploy on Grid'5000 at the end of November and the implementation has lasted **12 man-week**.It includes a learning phase (2 man-week), the optimization for caching dependencies, and the gathering of results in a central place (a CSV-like table with logs).

In total, the human effort is substantial. It is mainly due to the numerous "try and error" iterations needed and the difficulty of validating the infrastructure with regards to all

possible configurations. Moreover, the durations we reported concern the effort realized in the first place. Some modifications were also made in parallel to improve different parts of the solution – we count this duration in subsequent activities.

Obviously, this human cost is not absolute and other practitioners could have spent more or less time. We believe, however, that the engineering effort is not anecdotal and requires in any case a strong involvement and a multiplicity of expertises are needed to instrument the large-scale testing. The main difficulty behind this development was its error-proneness: we were often confronted to a try/error strategy, especially for the oracles, where finding the right error message (stacktraces) wasn't always so easy. We also constantly had the same reaction when a variant failed to build or compile: is the error coming from JHipster, is it a bug? or is our architecture causing the problem? or the machine wrongly configured?

This cost is not simply anecdotal: similar engineering effort is necessary needed for future versions of JHipster. We believe, however, that the human cost can decrease through the *reuse* of some components of our testing architecture. In an objective to work on more recent version of the generator, most of the current architecture is reusable. Only the variability model must be adapted, and possibly the "glue" for new technologies/frameworks for instance. The "glue" is special instructions we have to execute for certain technologies (the creation of some database for instance, or running specific Cassandra scripts).

RQ1.1: What is the cost of engineering an infrastructure capable of automatically deriving and testing all configurations?

*The testing infrastructure is itself a configurable system and requires a substantial engineering effort (**8 man-month**) to cover all design, implementation and validation activities, the latter being the most difficult.*

5.2. Computational cost

As we described in Chapter 4.5, we decided to use Grid'5000 to distribute the computing effort on multiple machines. This solution allowed us to test all 26,256 configurations in *less than a week*. Specifically, we performed a reservation of **80** machines for **4 periods** (4 nights) of **13 hours**.

The analysis of 6 configurations took on average about 60 minutes. This duration is consistent with the observations on our personal machines in which the processing of a JHipster configuration typically last 15 minutes with all the tests. The total execution of the workflow on the +26,000 configurations can then be achieved in about **4376 hour-machine** – 54.7 hours for 80 machines.

Once again, this cost (in time and machine) is specific to our solution (see Section 4.5.2). To reduce this overall execution time, however, we implemented few improvements such as the caching of the dependencies (Maven, Gradle and JavaScript) for instance (see Section 4.4.2).

Besides execution time, the processing of all variants also required a lot of disk space. Each scaffolded Web application occupies between 400MB and 450MB, thus forming a total of **5.2 terabytes**.

We replicated three times our exhaustive analysis (with minor modifications of our testing procedure each time); we found similar numbers for assessing the computational cost on Grid'5000. For instance, each reserved machine in Grid'5000 offers limited disk space preventing the treatment of a few hundreds of configurations. This limitation constrains us to restart these configurations.

As part of our last experiment, we observed suspicious failures for 2325 configurations with the same error message ("Communications link failure"). A new run of these configurations using additional machines yielded consistent results.

Overall, the computational cost is far beyond the testing budget of a software system. An exhaustive testing requires numerous individual machines with large disk space and CPU time. Even a small testing sample requires significant resources; it motivates our next research questions on the cost-effectiveness of sampling techniques.

RQ1.2: What are the computational resources needed to test all configurations?

Despite some optimizations (e.g., pre-caching of dependencies), testing all configurations requires a significant amount of computational resources
*(**4376 hour-machine** and **5,2To** of disk space).*

6. Faults and failures analysis

The execution of the derivation and testing workflow, presented in Chapter 4.4, yields a large CSV file comprising qualitative and quantitative data on all considered JHipster configurations. A refined version of this file is accessible on-line, at: `https://github.com/axel-halin/Thesis-JHipster/blob/master/Results/jhipster.csv`. This file allows to identify failing configurations, i.e., configurations that do not compile or build. In addition, we also exploited stack traces for grouping together some failures.

This result file and the grouping of the failures allows us to address **RQ2.1**. Moreover, we compare the efficiency of different sampling techniques in Section 6.4 to answer **RQ2.2**. Based on this comparison we determined the most cost-effective sampling technique (**RQ3**).

6.1. Preliminary failures analysis

Out of the **26,256** configurations we tested, we found that **34.37%** (i.e, **9376** configurations) failed. This failure occurred either at compile time (**224** configurations) or during build time (**9152**) although the generation was successful.

We took a look at the failing rate per feature to identify the most buggy configuration features. We present a subset of this analysis in this section (see Figure 6.1). Namely, we focus on the application type and the type of authentication. More details can be found in Appendix G (e.g. failing rate per database type, docker ...).

Regarding the application type feature, we observed that *microservice gateways* and *microservice applications*, with respectively **58.4**% (*4184*) and **58.3** (*532*) failing configurations, were more afflicted by failures than their counterparts, namely *monolithic applications* (**25.7**%, 4561) and *UAA servers* (**22.1**%, 99).

This disparity is also found in the authentication mechanism feature section. Out of the four different options, *JWT* and *HTTP Sessions* suffers from approximately the same percentage of failures: **20.5**% (*2279*) for the former; **22.3**% (*1586*) for the latter. *OAuth2* suffers from **39.7**% failing configurations (*1397*) while **UAA** authentication mechanism leads to **91.7**% (*4114*) failing configurations, the highest rate among all features.

This preliminary analysis, however, don't tell us exactly which features imply failures. Moreover, multiple failures can share the same fault.

We then conducted a more thorough statistical analysis to address **RQ 2.1**.

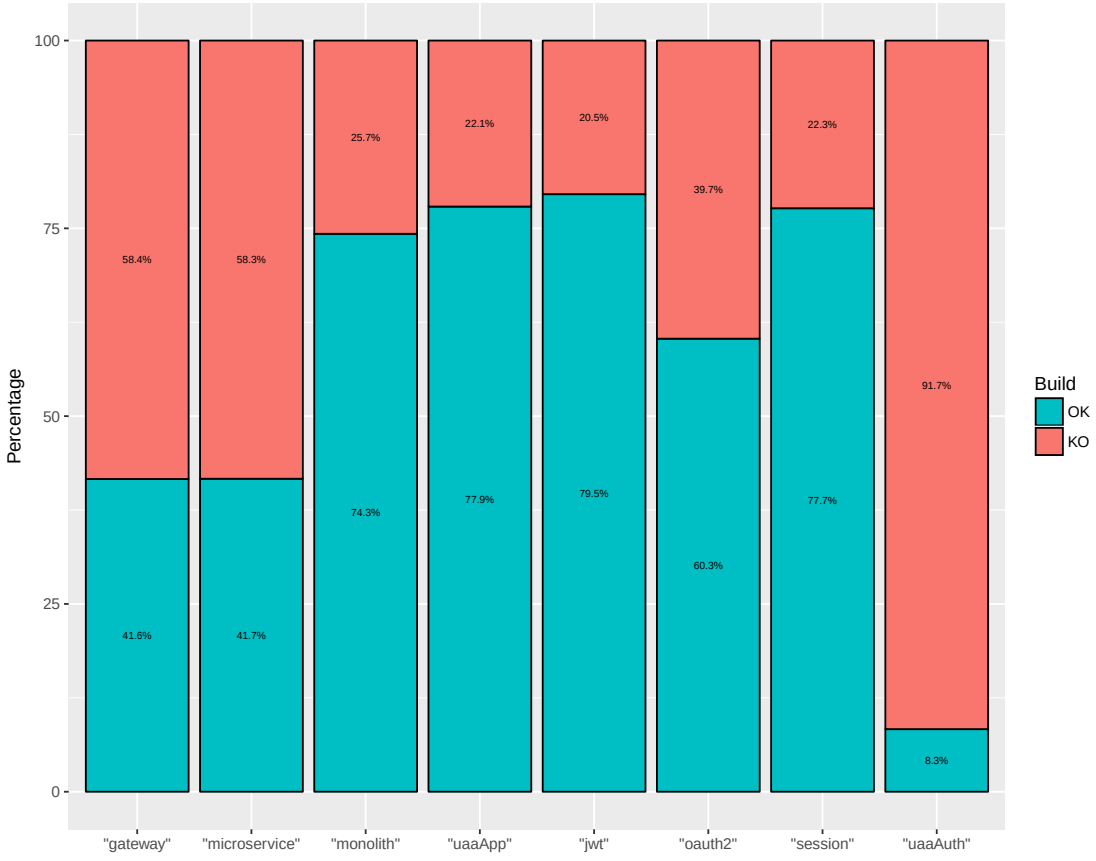

Figure 6.1.: Proportion of build failure by feature

6.2. Statistical analysis

Previous results, although enlightening about the most buggy features, do not show the root causes of the configuration failures – what features or interactions between features are involved in the failures?

6.2.1. Faults categorization

To investigate correlations between features and the failure results, we decided to use the Association Rule learning method (Hahsler et al., 2005). This method aims at extracting relations between variables of large data-sets and outputs a set of rules, each constituted by:

- *Left-hand side (LHS)*, the antecedent of the rule;
- *Right-hand side (RHS)*, the consequent of the rule;

- *Support*, the proportion of configurations where LHS holds;
- *Confidence*, the proportion of configurations where RHS holds.

We present, in Table 6.1, the rules extracted from the method which we parametrized as follows.

First, and as we focused on functional properties of the configurations, we restrained ourselves to rules where the RHS was either *Build=KO* or *Compile=KO*.

Second, we fixed the confidence to 1: if a rule has a confidence below 1 then it is not asserted in all configurations where the LHS expression holds – the failure does not occur in all cases.

Third, we lowered the support in order to catch all failures, even those afflicting smaller proportion of the configurations. For instance, only 224 configurations fail due to a compilation error; in spite a low support, we can still extract rules for which the RHS is *Compile=KO*.

We computed redundant rules using facilities of the R package *arules*[1]. As some association rules can contain already known constraints of the feature model (such as ***if*** *databaseType == MongoDB* ***then*** *devDatabaseType = MongoDB* ***and*** *prodDatabaseType = MongoDB*), we ignored some of them.

We first considered association rules for which the size of the LHS is either 1, 2 or 3. We extracted from the results 5 different rules involving two features (see Table 6.1). We found no rule involving 1 (all failures are then caused by an association of multiple features) or 3 features.

We then decided to have a look at the 200 association rules for which the LHS is of size 4 and found out a sixth association rule that incidentally corresponds to one of the first failures we encountered in the early stages of this study.

Moreover, as presented in Table 6.1, we only found one rule implicating a compilation failure. According to this rule, all configurations in which the database is MongoDB and social login feature is enabled (128 configurations) fail to compile. The other 5 rules are related to a build failure.

These association rules allow us to categorize the failures in 6 fault classes. These categories are presented in Section 6.3. The categorization explains **98.65**% of all failures (the combination of MariaDB and Gradle, alone, explains **37**% of all failing configurations, about **13**% of all configurations); the remaining failures have not yet been linked to a common fault. We do not know yet if these configurations are false-positives or are due to a undetected combination of features.

[1] `https://cran.r-project.org/web/packages/arules/index.html`

Left-hand side	Right-hand side	Support	Confidence	Github Issue	Report/Correction date
DatabaseType="mongodb", EnableSocialSignIn=true	Compile=KO	0.488 %	1	4037	27 Aug 2016 (report and fix for milestone 3.7.0)
prodDatabaseType="mariadb", buildTool="gradle"	Build=KO	16.179 %	1	4222	27 Sep 2016 (report and fix for milestone 3.9.0)
Docker=true, authenticationType="uaa"	Build=KO	6.825 %	1	UAA is in Beta	Not corrected
authenticationType="uaa", hibernateCache = "no"	Build=KO	2.438 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
authenticationType="uaa", hibernateCache = "ehcache"	Build=KO	2.194 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
prodDatabaseType="mariadb", applicationType = "monolith", searchEngine = "false", Docker = "true"	Build=KO	5.59%	1	4543	24 Nov 2016 (report and fix for milestone 3.12.0)

Table 6.1.: Association rules involving compilation and build failures

6.2.2. Association rules overlapping

The Association rule learning method, however, suffers from a couple of drawbacks.

Besides the fact that it doesn't take into account the constraints of the feature model, as previously mentioned, we also observed an overlapping between the different rules.

Let's consider the fault we mentioned here-before: the combination of *MariaDB* and *Gradle*. As we mentioned, this bug covers 13% of the configuration space. However, the association rule extracted from the method (second line of Table 6.1) specifies a support of 16.179%.

This difference is due to a rule overlapping. As presented in the next section, we identify 2 fault classes related to the use of MariaDB: once when used with Docker and once without it. As depicted in Table 6.2, these two configurations have different stacktraces.

The problem is that the method takes the most general rule, in our case it outputs *MariaDB and Gradle implies Build failure*. This rule covers the second fault and part of the first fault (see next section). The latter comprises configurations relying on Maven and on Gradle (the two available build tools).

JHipsterRegister	Docker	prodDatabaseType	buildTool	Build	Log-Build
jhipster68	true	"mariadb"	"gradle"	KO	Error parsing reference: ""jhipster - jhipster-mariadb"" is not a valid repository/tag
jhipster68	false	"mariadb"	"gradle"	KO	Failed to get driver instance for jdbcUrl= jdbc:mariadb://localhost:3306/jhipster

Table 6.2.: CSV file excerpt

To overcome this limitation and to classify the failures in fault classes, we relied on the stacktraces and error messages extracted during the execution of the workflow. We identified one pattern per fault.

This stacktraces analysis allowed us to characterize the proportion of configuration impacted by a specific fault. These proportions are presented in Figure 6.2.

Despite these drawbacks, the association rule learning method remains useful to identify combination of features leading to a failure.

6.3. Qualitative analysis

We now summarize and characterize the 6 important faults classes we identified. These faults are caused by the interaction of 2 or 4 features and are responsible of **34.37%** failing configurations. More details can be found in Appendix F. We present in Figure 6.2 the distribution of all failures among the different fault classes.

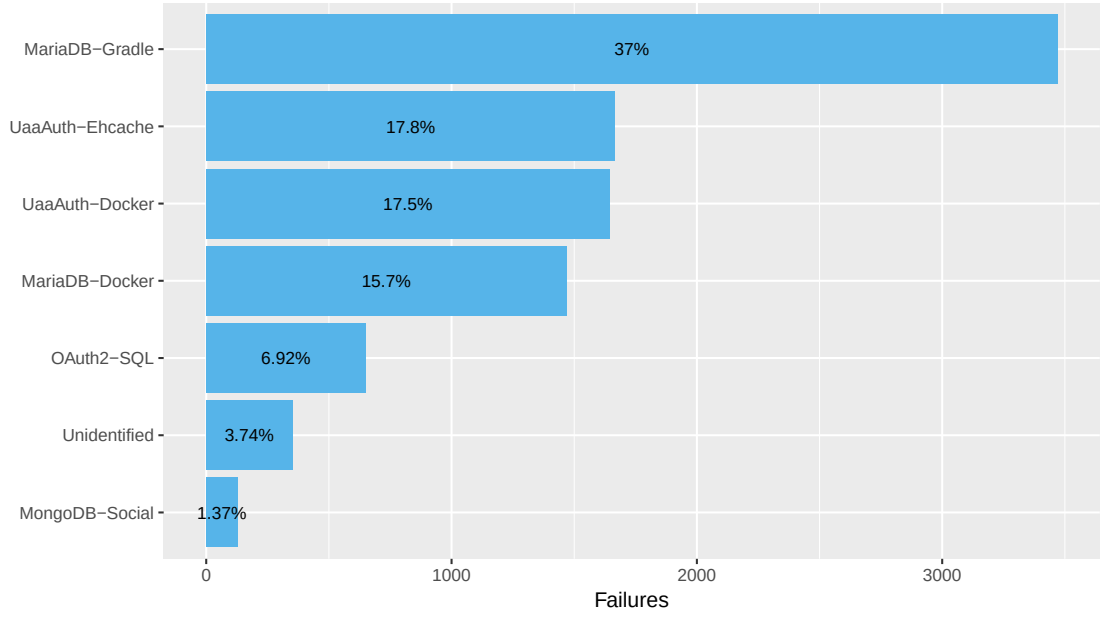

Figure 6.2.: Proportion of failures by fault

#1 MariaDB with Docker: This fault is the only one caused by the interaction of 4 features: it concerns *monolithic* web-applications relying on MariaDB as production database, where the search-engine (ElasticSearch) is disabled and built with Docker. These variants amount to **1468** configurations and the root cause of this bug lies in the template file *src/main/docker/_app.yml* where a condition (*if prodDB = MariaDB*) is missing.

#2 MariaDB using Gradle: This second fault concerns variants relying on Gradle as build tool and MariaDB as the database (**3469** configurations). It is caused by a missing dependency in template file *server/template/gradle/_liquibase.gradle*.

#3 UAA authentication with Docker: The third fault occurs in Microservice Gateways or Microservice applications using an UAA server as authentication mechanism (**1644** Web apps). This bug is encountered at build time, with Docker, and it is due to the absence of UAA server Docker image. It is a known issue but it has not been corrected yet, UAA servers are still in beta versions.

#4 UAA authentication with Ehcache as Hibernate 2nd level cache: This fourth fault concerns Microservice Gateways and Microservice applications, using a UAA authentication mechanism. When deploying manually (i.e., with Maven or Gradle), the web application is unable to reach the deployed UAA instance. This bug seems to be related to the selection of Hibernate cache and impacts **1667** configurations.

#5 OAuth2 authentication with SQL database: This defect is faced **649** times, when trying to deploy a web-app, using a SQL database (Mysql, PostgreSQL or MariaDB) and an OAuth2 authentication, with Docker. It was reported on August 20th, 2016 but the JHipster team was unable to reproduce it on their end.

#6 Social Login with MongoDB: This sixth fault is the only one occurring at compile time. Combining MongoDB and social login leads to **128** configurations that fail. The source of this issue is a missing import in class *SocialUserConnection.java*. This import is not in a conditional compilation condition in the template file while it should be.

#7 Unidentified failures: We also found **351** failures for which a common fault was not identified. After a more thorough analysis, we found that these were mostly false-positives and that the failures were caused by several different faults among which failure of dependencies fetching due to connection issues, database related scripts issues (in Cassandra configurations) or trouble in the workflow execution (server not killed, database still running, etc.)

RQ2.1: How many and what kinds of failures/faults can be found in all configurations?

*Exhaustive testing shows that **34.37%** of the configurations fail. Our analysis identifies **6 interaction faults** (caused by the interaction of **2 or 4 features**) as the root cause for this high percentage.*

6.4. Sampling techniques comparison

As we have previously shown (see Section 5.2) the testing of all configurations is very costly. Sampling techniques thus remain of interest. Ideally, we would like to find a

maximum of failures and faults with a minimum of configurations in the sampling.

We selected and applied numerous sampling techniques considered in the literature. For each technique, we report on the number of faults and failures identified (we consider, here, that it is sufficient to find one failure for identifying the associated fault). We then compare their effectiveness to address **RQ2.2**. We summarize our results in Table 6.3.

6.4.1. Sampling techniques

Combinatorial Interaction Testing

The first sampling technique we considered is based on combinatorial interaction testing (Cohen et al., 2008; Mathur, 2008). We focused on the **t-wise** criteria. This sampling technique selects a subset of the configuration space such as each interaction of T features is covered (e.g., (Perrouin et al., 2010; Cohen et al., 2008; Yilmaz et al., 2006)). We selected 4 variations of the criteria: **1-wise**, **2-wise**, **3-wise** and **4-wise**. We generated the matching samples with *SPLCAT* (Johansen, Haugen, & Fleurey, 2012).

The 4 variations yield samples of respectively **8**, **41**, **126** and **374** configurations. **1-wise** only finds **2** faults; **2-wise** discovers **5 out of 6 faults**; **3-wise** and **4-wise** find **all** of them. Regarding the proportion of failures, the more the sample size, the more failures we found (from **2** failures among **8** configurations for **1-wise** to **161** on **374** configurations for **4-wise**).

Sampling technique	Sample size	Failures	σ of Failures	Failures efficiency	Faults	σ of Faults	Fault efficiency
Random(8)	8	2.8568	1.3133	34.37%	2.18	0.9784	27.25%
PLEDGE(8)	8	3.16	1.2303	39.50%	2.14	0.8245	26.75%
1-wise	8	2	/	25.00%	2	/	25.00%
Random(12)	12	4.2852	1.7895	34.37%	2.7	1.0396	22.5%
PLEDGE(12)	12	4.92	1.2303	41.00%	2.82	0.9087	23.50%
2-wise	41	14	/	34.15%	5	/	12.20%
Random(41)	41	14.6411	3.1823	34.37%	4.49	0.7177	10.95%
PLEDGE(41)	41	17.64	2.5041	43.02%	4.70	0.8306	11.46%
3-wise	126	52	/	41.27%	6	/	4.76%
Random(126)	126	44.9946	4.9114	34.37%	5.28	0.5333	4.19%
PLEDGE(126)	126	49.08	11.5807	38.95%	4.66	0.6977	3.70%
Random(374)	374	133.5554	8.4064	34.37%	5.58	0.4960	1.49%
PLEDGE(374)	374	139.20	31.7975	37.17%	4.62	1.1814	1.24%
4-wise	374	161	/	43.05%	6	/	1.60%
Most-enabled-disabled	574	190	/	33.10%	2	/	0.35%
One-disabled	922	253	/	27.44%	5	/	0.54%
One-enabled	2,340	872	/	37.26%	6	/	0.26%
ALL	26,256	9,376	/	34.37%	6	/	0.02%

Table 6.3.: Efficiency of different sampling techniques

One-disabled

The second sampling technique we implemented is based on the so-called **one-disabled** algorithm (Abal et al., 2014). The idea of this algorithm is to deactivate all features except one, and this for all features.

Specifically, in our implementation, we consider all *optional* features that are direct children of the root and those that do not have sub-features (ElasticSearch) in the feature model of Figure 4.3. Other optional features (such as HibernateCache) that are included under more complex conditions (e.g., the inclusion of parent features that are also optional) are not taken into account.

Moreover, by selecting only a subset of all features we are confronted to many suitable configurations (i.e, sharing the same subset of features) and thus many optimal[2] solutions. When confronted to several optimal solutions we select all of them rather the first one, as it is proposed in (Medeiros et al., 2016).

The selected features are involved in the top questions of the JHipster configurator; the hope is to find numerous defects with their inclusions. Technically, we retain all valid configurations where one optional feature is disabled and the other optionals are enabled. (there are 8 possible combinations).

For instance, we can select each configuration where *Docker* is disabled and *Spring-WebSockets*, *InternationalizationSupport*, *SocialLogin*, *Libsass* and *ClusteredSession* are enabled; the process is then repeated for all other combinations of optional features.

Practically, we select the configurations of the samples on the basis of the CSV results file. Hence, all selected configurations are valid with regards to the constraints of the feature model.

This sampling technique yields a total sample of **922 configurations** among which **253 failures** (22.44% of the sample). This technique identifies **all** major faults **but one**.

One-enabled and Most-enabled-disabled

Third, we implemented two variations of the previous algorithm: **one-enabled** and **most-enabled-disabled** algorithms (Abal et al., 2014; Medeiros et al., 2016).

The former mirrors *one-disabled* and consists in enabling each optional feature one at a time. The latter selects all configurations where (1) most of the optional features are selected and (2) most of the optional features are deselected.

The samples generation was similar to the one of *one-disabled* algorithm: the configurations are then valid with regards to the feature model.

[2]The optimality of a configuration in this context is based on the number of enabled features. The configuration having the largest set of enabled features is the optimal solution.

One-enabled extracts a sample of **2340 configurations** with **872 failures** (37.26%) and identifies every major fault. Most-enabled-disabled gives a sample of **574 configurations** comprising **190 failures** (33.10%) and only identifies **2 faults**.

Dissimilarity

Fourth, we also considered *dissimilarity* testing for Software Product Lines (Henard et al., 2014; Al-Hajjaji et al., 2016), using a tool called **PLEDGE** (Henard et al., 2013).

This technique approximates t-wise coverage by generating dissimilar configurations (in terms of shared features amongst these configurations). From a set of random configurations of a specified cardinality, an evolutionary algorithm evolves this set such that the distances amongst configuration are maximal – by replacing a configuration at each iteration – within a certain amount of time. We retained this technique because it can afford any testing budget (sample size and generation time).

For each sample size, we report the average failures and faults for 100 PLEDGE executions with the Greedy method in 60 secs (Henard et al., 2013). In Table 6.3, we present results for key size samples (respectively **8**, **12**, **41**, **126** and **374** configurations) for comparing with other sampling techniques. We find, on average, respectively **2**, **3**, **5** and **5** faults.

Random

Finally, we considered **random** samples from size 1 to 2500.

The random samples exhibit, by construction, 34.37% of failures (the same percentage that is in the whole dataset). To compute the number of unique faults, we simulated 100 random selections. Along with the mean of defects found we present the standard deviation value (σ).

We find, on average, respectively **2.18**, **2.7**, **4.49**, **5.28** and **5.58** faults for respectively **8**, **12**, **41**, **126** and **374** configurations.

As we can see in Figure 6.4, we need to run $\approx$ 2000 configurations to find *all* the faults. It's due to the sixth fault (Social Login with MongoDB) which is hard to find with the random technique due to the less representation of this fault (1.37 %).

6.4.2. Sampling techniques comparison

In order to compare the efficiency of the different sampling techniques (i.e, their ability to find fault and failures), we considered two main metrics taking into account the sample size.

Failure efficiency is the ratio of *failures to sample size*; similarly, **Fault efficiency** is the ratio of *faults to sample size*.

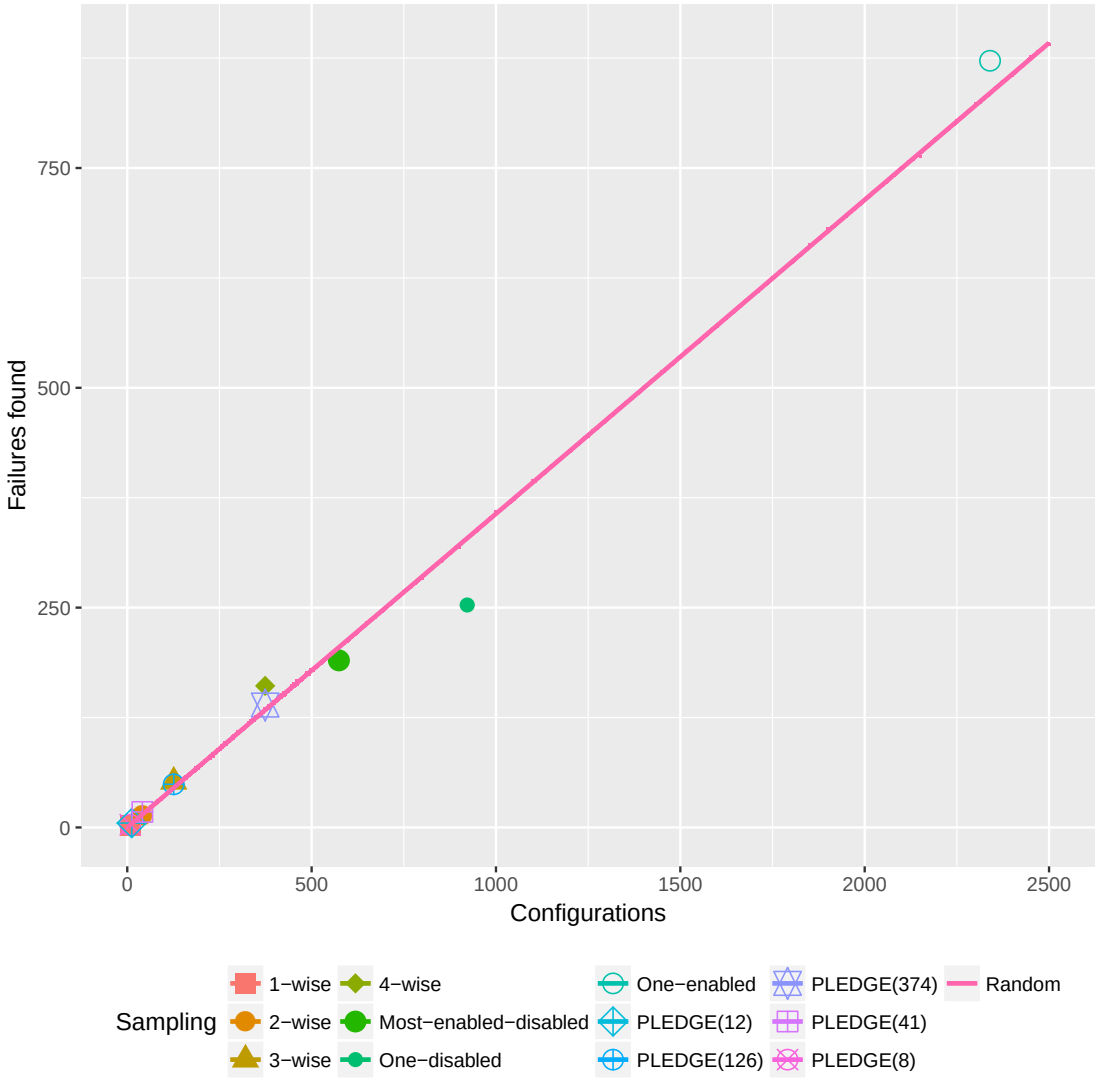

Figure 6.3.: Failures found by sampling techniques

The results are presented in Table 6.3. Figure 6.3 (respectively, Figure 6.4) presents the evolution of *failures* (respectively, *faults*) with regards to the sample size. We consider on these figures the sampling techniques presented in Table 6.3 and random sample of size *1 to 2500*.

A *first observation* is that random is a strong baseline for both failures and faults. It is, however, more efficient to discover failures than faults. Indeed, and quite logically, the more the configurations, the more failures are found. Nonetheless, it takes a random set of *2500* configurations to get a mean value of 6 detected faults. 2-wise or 3-wise sampling techniques are slightly more efficient to identify faults than random. The significant

difference between these two t-wise variation is explained by the sample size: although the latter finds all the bugs (one more than 2-wise) its sample size is triple (126 configurations against 41 for 2-wise). 1-wise has the best fault efficiency: in ensuring that each feature is present at least once in the sample, it selected 2 combinations of features leading to a fault. On the contrary, one-enabled, one-disabled and most-enabled-disabled identify less faults than random samples of the same size. PLEDGE is superior to random for small sample sizes. In general, a relatively small sample is sufficient to quickly identify the 5 or 6 most important faults – there is no need to cover the whole configuration space.

A *second observation* is that there is no correlation between failure efficiency and fault efficiency. For example, one-enabled has a failure efficiency of 37.26% (better than Random and many techniques) but is one of the worst technique in terms of fault rate due of its high sample size. In addition, some techniques, like most-enable-disabled, can find numerous failures that in fact correspond to the same fault.

Moreover, our results show that the choice of a metric (failure-detection or fault-detection capability) can largely influence the choice of a sampling technique.

Our initial assumption was that the detection of one failure leads to the finding of the associated fault. It is the case in JHipster: contributors can easily find the root causes based on a *manual* analysis of a failure. On our side, we can group together failures having the same stacktraces.

However, this assumption may not hold in other contexts. That is, finding the feature interaction faults can be much more difficult and less immediate. In this case, it is beneficial to *replicate* a fault with many failures. We can use statistical methods and augment the *support* of failed configurations to identify feature interaction faults. As a result, the ability of finding failures may be quite important. A tradeoff between failure and fault efficiency can certainly be considered.

In our case study, 3-wise is a good candidate since all faults are found and its failure efficiency is superior to most of the techniques.

RQ2.2: How effective are sampling techniques comparatively?

- random is a strong baseline for both failures and faults;
- one-enabled, one-disabled and most-enabled-disabled identify less faults than random samples of the same size;
- PLEDGE is superior to random for small sample sizes;
- 2-wise or 3-wise sampling techniques are slightly more efficient to identify faults than random;
- 1-wise has the best fault efficiency.

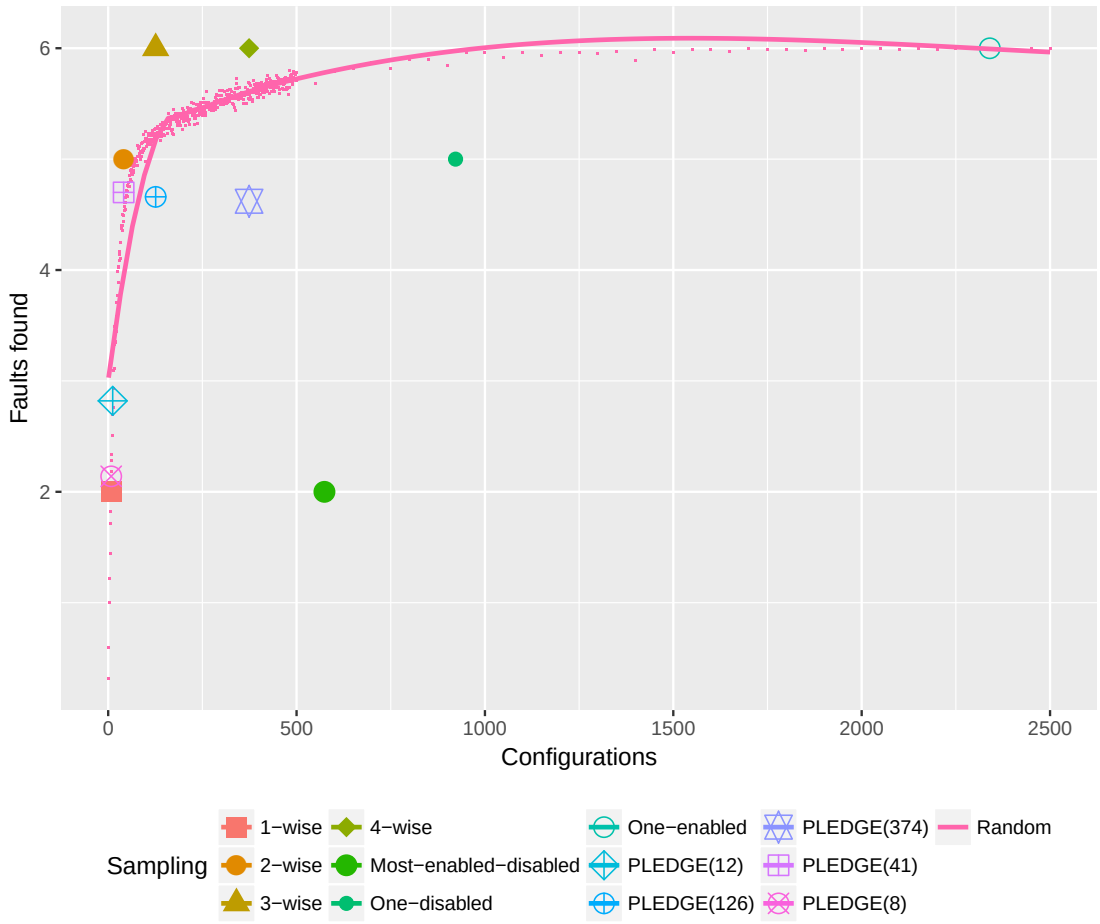

Figure 6.4.: Faults found by sampling techniques

7. Practitioners viewpoint

At the end of January, we interviewed the JHipster lead developer, *Julien Dubois*, during 1.5 hour. We prepared a set of questions and performed a semi-structured interview on Skype for allowing new ideas during the meeting. We then exchanged emails with two core developers of JHipster, *Deepu K. Sasidharan* and *Pascal Grimaud*.

Based on an early draft of a second paper we submitted, they clarified some points and freely reacted to some of our recommendations. We wanted to get insights on how JHipster was developed, used, and tested. Our goal was to validate some assumptions of our work and to confront our empirical results with their current practice.

7.1. Jhipster's testing strategy

To test their generator, the JHipster team relies on a continuous integration platform, *Travis*[1], integrated into GitHub.

At the time of the release 3.6.1, the free installation of Travis allowed to perform 5 different builds in parallel, at each commit. The team decided to use this feature on only 12 different configurations.

During our discussion they provided the following explanations: *"The only limit was that you can only run 5 concurrent jobs so having more options would take more time to run the CI and hence affect our turn around hence we decided on a practical limit on the number [...] We only test the 12 combinations because we focus on most popular options and leave the less popular options out."*

Besides this facility, Julien Dubois also mentioned that his company (*Ippon Technologies*) provides some machines used to perform additional tests. We can consider that the testing budget of JHipster 3.6.1 was limited to 12 configurations.

This has a strong implication on our empirical results: *despite their effectiveness, some sampling strategies we have considered exceed the available testing budget of the project.* For example, a 2-wise sample has 41 configurations and is not adequate. A remaining solution is dissimilarity sampling (PLEDGE) of 12 configurations, capable of finding 5 failures and 3 faults.

Moreover, the selection of these 12 configurations is also interesting.

[1] `https://travis-ci.org/`

According to Julien Dubois, it is both based on (1) intimate technical knowledge of the technologies and (2) a statistical prioritization approach. Specifically, when a given JHipster installation is configured, the user can send anonymous data to the the JHipster team so that it is possible to obtain a partial view on the configurations installed. The most popular features have been retained to choose the 12 configurations.

For example, this may partly explain that configurations with Gradle are more buggy than those with Maven – we learned that Gradle is used in less than 20% of installations. There were also some discussions about improving the maintenance of Gradle, due to its popularity within a subset of contributors.

The prioritization of popular configurations is perfectly understandable. Such a sample has the merit of ensuring that, at each commit, popular combinations of features are still valid (acting as non-regression tests). However, corner cases and some feature interactions are not covered, possibly leading to high percentage of failures.

Julien Dubois also noted that many developers fix configurations themselves without always filling them in the bug tracker. The generator is also used by many companies who probably also fix bugs by themselves without necessarily reporting the issues.

7.2. Merits and limits of exhaustive testing

Julien Dubois welcomed the initiative and was seriously impressed by the unprecedented engineering effort and the 34.37% failures. We asked whether the version 3.6.1 had special properties, perhaps explaining the 34.37% of failures but he refuted this assumption and rather stated that the JHipster version was a major and stable release.

We explained that most of the defects we found were reported by the JHipster community. The lead developer was aware of some interactions that caused problems in JHipster. These are known mostly from experience and not via the application of a systematic process. However, he ignored the significance of the failures.

The high percentage of failures we found should be seriously considered since a significant number of users may be impacted given the popularity of the project. Even if faults involve rarely used configurations, he considered that the strength of JHipster is precisely to offer a diverse set of technologies.

The effort of finding many failures and faults is therefore highly valuable.

We then discussed the limits of testing all configurations. The cost of building a grid/cluster infrastructure is currently out of reach for the JHipster open-source project, due to the current lack of investments.

JHipster developers stated that *"even if we have limitless testing infrastructure, I do not think we will ever test out all possible options due to the time it would take."*

This observation is not in contradiction with our research method. Our goal was not to promote an exhaustive testing of JHipster but rather to investigate a cost-effective strategy based on collected evidence.

Another important insight is that *"the testing budget was more based on the time it would take and the resource it would use on a free infrastructure. If we let each continuous integration build to run for few hours then we would have to wait that long to merge pull request and to make releases etc. So it adds up lag affecting our ability to release quickly and add features and fixes quickly. So turn around IMO is something you need to consider for continuous integration."*

Finally, Julien Dubois mentioned an initiative[2] to build an all-inclusive environment capable of hosting any configuration. It is for JHipster developers and aims to ease the testing of a JHipster configuration on a local machine. In our case, we have build a similar environment with the additional objective of automating the test of configurations. We have also validated this environment on a large scale.

7.3. Discussion

On the basis of multiple collected insights, we emitted several recommendations regarding the testing of the configurator. We discuss here-after the different trade-offs to consider when testing JHipster and address **RQ3**.

The first recommendation we made regarded the sampling strategy.

Our empirical results suggest to use a dissimilarity sampling strategy in replacement to the current sampling based on statistical prioritization. It is one of the most effective strategy for finding failures and faults and it does not exceed the budget. In general, the focus should be on covering as much as possible feature interactions. If the testing budget can be sufficiently increased, t-wise strategies can be considered as well. Random strategies are less effective both in terms of faults and failures efficiency contrarily to the study of Medeiros et al (Medeiros et al., 2016). Thus the efficiency debate between random sampling and combinatorial interaction testing is not over (e.g., (Arcuri & Briand, 2012)). Random, nevertheless, has the advantage of not requiring a Feature model to be deployed.

However, developers remind us that *"from a practical standpoint, a random sampling has possibility of us missing an issue in a very popular option thus causing huge impact, forcing us to make emergency releases etc, where as missing issues in a rarely used option does not have that implication"*. This applies to t-wise and dissimilarity techniques as well. Hence, one should find a trade-off between cost, popularity, and effectiveness of sampling techniques (for instance, a mix of random and popular configurations can be

[2] `https://github.com/jhipster/jhipster-devbox`

a strategy). We see this as an opportunity to further experiment with multi-objective techniques (Sayyad et al., 2013; Parejo et al., 2016; Le Traon, 2015).

The second recommendation regarded the size of the considered sample.

Our empirical results and discussions with JHipster developers suggest that the testing budget was simply too low for JHipster 3.6.1, especially when popular configurations are included in the sampling. According to JHipster developers, the testing budget *"has increased to 19 now with JHipster 4, and we also have additional batch jobs running daily tripling the number of combinations [...] We settled on 19 configurations to keep build times within acceptable limits. Discussions are here* `https://github.com/jhipster/generator-jhipster/issues/4301` *"*

An ambitious and long-term objective is to crowdsource the testing effort with contributors. Users can lend their machines for testing some JHipster configurations while a subset of developers could also be involved with the help of dedicated machines. In complement to continuous testing of some popular configurations, a parallel effort could be made to seek failures (if any) on a diversified set of configurations, possibly less popular.

Finally, the last recommendation we emitted concerned the development and *maintenance* of a configuration-aware testing infrastructure. Without a ready-to-use environment, contributors will not be able to help in testing configurations. It is also pointless to increase the sample if there is no automated procedure capable of processing the constituted configurations.

The major challenge will be to follow the evolution of JHipster and make the testing tractable. A formal model of the configurator should be extracted for logically reasoning and implementing random or t-wise sampling. New or modified features of JHipster should be handled in the testing workflow; they can also have an impact on the tools and packages needed to instrument the process.

RQ3: What is the most cost-effective sampling strategy?

Dissimilarity and t-wise sampling are the most effective but there is no correlation between failure and fault efficiencies. However, only one variation of t-wise (1-wise) and dissimilarity are within reach of JHipster testing budget

8. Threats to validity

We however identified several threats to the validity of our study which we regroup in two categories: *external validity* and *internal validity*.

External validity

Our engineering effort has focused on a single but industrial-strength and complex system. We expect more insights into characteristics of real-world systems than using diverse but smaller or synthetic benchmarks.

The JHipster case allows us to envision, for the first time, the exhaustive testing of a large configurable system. With a ground truth, we can assess sampling techniques with a high confidence. The multiple collected evidence contributes to the body of knowledge established on other configurable systems.

For increasing external validity, we hope to replicate our work on new versions of JHipster. This, however, will require substantial resources and engineering effort.

Internal validity

Threats to internal validity are mainly related to the quality of our testing infrastructure.

An error in the feature model or in the configuration-aware testing workflow can typically produce wrong failures. As reported, the validation of our solution has been a major concern during 8 man-month of development. We have used several strategies, from statistical computations to manual reviews of individual failures to mitigate this threat. We found all faults reported by the JHipster community and new failures.

For the other remaining 351 failures, there might be false positives (as explained in Section 6.3) since we have not looked at each individual failure. It only represents **1.33**% (See unidentified failures in Figure 6.2) of JHipster configurations and would have a marginal incidence on the results.

Part V.

Conclusion

9. Conclusion and perspectives

We now conclude this thesis by summarizing the different contributions and results and by presenting a few perspectives and future work possibilities.

9.1. Conclusion

In this thesis, we first contextualized our study by presenting the notions of Software Product Line, which aims at achieving mass customization, and variability. We also introduced the concept of variability-intensive systems.

We briefly reviewed the state of the art regarding the testing of such systems and in particular the use of sampling techniques.

We then introduced our case-study: JHipster, an open-source Yeoman generator of Web-applications. We argued it presented itself as an interesting case study due to its diversity in term of technologies and its manageability in term of configurations (**162,508**). This reasonable number allow us to exhaustively testing Jhipster and create a ground truth with which we can assess the error-detection capabilities of sampling techniques. Then, we briefly described its internal and external behaviour.

We argued that JHipster is an interesting case studies for many research directions and that the problems we address here (See the following research questions) are common to most configurable systems.

We then formalized the following research questions:

- **(RQ1.1)** What is the cost of engineering an infrastructure capable of automatically deriving and testing all configurations?
- **(RQ1.2)** What are the computational resources needed to test all configurations?
- **(RQ2.1)** How many and what kinds of failures/faults can be found in all configurations?
- **(RQ2.2)** How effective are sampling techniques comparatively?
- **(RQ3)** What is the most cost-effective sampling strategy?

Next, we described the engineering effort necessary to address **RQ1**. This engineering effort consists in 4 steps: the modelling of JHipster variability, the enumeration of all valid configurations (with regard to the variability model), the development of the testing infrastructure and the development of a testing environment. We characterized for each step its cost (**RQ1.1**) for a total of (**8 man-month**) engineering efforts. We elicited the resources needed to run the workflow on all configurations (**RQ1.2**) with a significant amount of computational resources (**4376 hour-machine** and **5,2To** of disk space), despite some optimizations.

We then presented our analysis of the data extracted from this workflow execution. We found that about **34.37%** of the configurations failed to deploy and that some features were more afflicted by faults than others (for instance, UAA-based microservices show a failure rate of 91.66%). Going further we found that most of the failures were caused by **6 faults**, due to a combination of **2 or 4 features** (**RQ2.1**).

Finally, we compare the efficiency of an ALL-testing strategy to different, state of the art, sampling techniques (**RQ2.2**). We selected 4 variations of the t-wise criteria, random samples and 3 other sampling technique (namely, One-Enabled, One-Disabled and Most-Enabled-Disabled). We also compare these results with sample obtained with prioritization algorithm (with PLEDGE). On the basis of this comparison we concluded that the most cost-effective sampling strategies were dissimilarity and t-wise (**RQ3**).

We also confronted our results with some core developers of JHipster. We conducted a 1,5h semi-structured interview with Julien Dubois, lead developer of JHipster, and exchanged several emails with Deepu Sasidharan and Pascal Grimaud. We presented them a few recommendation and discussed their feasibility.

Beyond this study, JHipster can prove to be an interesting case study for different research directions. We present some of them in the following Section.

9.2. Perspectives

This work (and especially the result files) may also be the basis on which other studies could be performed. In (Halin et al., 2017), we offered several research directions for which we believe JHipster can be a good case-study. We present in this Chapter only a few possibilities to extend this study, for the sake of conciseness.

Replicating the study

The first study that could be performed is the replication of the experiment presented in this thesis on other JHipster version. This replication would be relevant for both previous versions but also more recent ones.

For starter, it could help detect potential faults in the latest version of the generator and offer the JHipster community a gain of several weeks or month in the fixing of deviant configurations (as mentioned in Section 6.3, a bug afflicting JHipster 3.6.1 was corrected 2 months after the release).

Furthermore, the replication of the study on several versions of the generator could provide an overview of its evolution and more specifically the correction of faults. Depending on the results it could strengthen our observations regarding JHipster testing strategy (in early stages of JHipster development, how long did it take to find and correct faults?).

The method and source code used to assess JHipster 3.6.1 can be reused on other versions (previous and later). A few adaptations might be necessary depending on new frameworks/technologies. First, the modelling of the variability should be updated. New added frameworks might require some "glue", quite similarly to the database related one (starting services and so on). Last but not least, the Debian image might need some updates for the configurations to execute correctly. These two last points would be more difficult than the first to set up as it might require try/error strategy (e.g., executing one configuration and seeing if any package that should be installed is missing).

Community-driven configuration testing

Another motivation is to maintain and develop the current testing infrastructure. Our goal would be to delegate the maintenance effort to the JHipster community.

This, however, would require two steps:

- the development (and maintenance) of an all-inclusive testing infrastructure (similar to ours);
- the delegation of the testing effort across several nodes.

A step has already been made in that direction with the development of the JHipster DevBox[1] (this concurrent project currently serves as a testing environment for the core developers to run tests locally and assess some configurations).

An alternative could be to rely on configuration management tools such as Ansible[2].

Furthermore, as we previously showed, the thorough testing of all configurations is infeasible on a unique machine. One could then imagine a crowd-sourcing initiative to evaluate JHipster variants and alleviate the core JHipster team of the testing phase.

[1] `https://github.com/jhipster/jhipster-devbox`
[2] `https://www.ansible.com/`

Non-functional analysis

In this study, we focused on functional properties of JHipster variants (generation, compilation, build). Nevertheless, in the development of the analysis workflow, we have included several steps yielding qualitative data regarding, for instance, the performance of a web application. Although we didn't have the time to analyse these data, there are still present in the CSV file.

A future work would then be to take advantage of the data to draw conclusions of non-functional properties of JHipster variants and extend our notion of deviant behaviour to more than just functional properties.

Extend to other case studies

Finally, we mentioned in Section 3.5 that the issues considered here (the testing of all configurations, finding the most adapted sampling techniques, etc.) are common to most configurable systems.

A future work could be to replicate this study on other real-life case studies to compare the results and recommendations regarding the testing strategy to adopt.

Variability-aware static analysis

In this study, we decided to automatically derived all configuration in order to assess them one at a time. Another research direction, common in the state of the art, would be to rely on variability-aware static analysis to detect faults.

This would reduce the overall cost of testing but it is also not so trivial to set up due to the variety of technologies at hand in JHipster (Java, XML, JavaScript, *etc.*)

One-enabled, one-disabled, most-enabled-disabled

Part of what is presented in this master thesis (other than what was presented in VaMoS'17) has been the subject of a submission to an international conference. The feedbacks about our methodology and results were received too late to all be included here. We however discuss one update we propose to work on soon.

As presented in Section 6.4.1, our implementation of *one-enabled*, *one-disabled* and *most-enabled-disabled* vary from the one offered by the authors themselves. Due to some remarks we received, we propose to implement these sampling algorithm as it is specified in (Medeiros et al., 2016).

This new implementation would be added next to the one we already have (and that is relevant) and would not aim at replacing it. This would then leave our results and observations untouched.

References

Abal, I., Brabrand, C., & Wasowski, A. (2014). 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th acm/ieee international conference on automated software engineering* (pp. 421–432). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2642937.2642990 doi: 10.1145/2642937.2642990

Acher, M., Baudry, B., Heymans, P., Cleve, A., & Hainaut, J.-L. (2013). Support for reverse engineering and maintaining feature models. *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, 20. Retrieved from http://dl.acm.org/citation.cfm?id=2430502.2430530{%}5Cnhttps://hal.inria.fr/hal-00766786/document doi: 10.1145/2430502.2430530

Acher, M., Collet, P., Lahire, P., & France, R. B. (2011). Managing feature models with familiar: a demonstration of the language and its tool support. *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings*, 91–96. Retrieved from http://doi.acm.org/10.1145/1944892.1944903 doi: 10.1145/1944892.1944903

Acher, M., Collet, P., Lahire, P., & France, R. B. (2013). Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, *78*(6), 657-681.

Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., & Saake, G. (2016). IncLing: efficient product-line testing using incremental pairwise sampling. In *Gpce '16* (pp. 144–155). ACM.

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). *Feature-oriented software product lines: Concepts and implementation*. Springer Berlin Heidelberg. Retrieved from https://books.google.be/books?id=4WUnAQAAQBAJ

Arcuri, A., & Briand, L. (2012, Sept). Formal analysis of the probability of interaction fault detection using random testing. *IEEE Transactions on Software Engineering*, *38*(5), 1088-1099. doi: 10.1109/TSE.2011.85

Bachmann, F., & Clements, P. C. (2005). Variability in Software Product Lines. (CMU/SEI-2005-TR-012), 1–13. Retrieved from http://www.sei.cmu.edu/library/abstracts/reports/05tr012.cfm

Balouek, D., Carpen-amarie, A., Charrier, G., Jeannot, E., Jeanvoine, E., Adrien, L., ... Nussbaum, L. (2012). Adding Virtualization Capabilities to Grid ' 5000. , 18.

Beuche, D., Papajewski, H., & Schröder-Preikschat, W. (2004). Variability management with feature models. *Science of Computer Programming*, *53*(3), 333–352. doi: 10.1016/j.scico.2003.04.005

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., & Pohl, K. (2002). Variability Issues in Software Product Lines. *Software Product Family Engineering*, *2290*, 13–21. Retrieved from `http://www.springerlink.com/index/TPW4XC6TL8K345DT.pdf` doi: 10.1007/3-540-47833-7_3

Boucher, Q., Heymans, P., Abbasi, E. K., Hubaux, A., & Acher, M. (2013). What's in a Web Configurator ? Empirical Results from 111 Cases. , 1–20.

Chen, L., Babar, M. A., & Ali, N. (2009). Variability Management in Software Product Lines : A Systematic Review. *SPLC '09 Proceedings of the 13th International Software Product Line Conference*, Pages 81–90.

Classen, A., Boucher, Q., & Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, *76*(12), 1130–1143. doi: 10.1016/j.scico.2010.10.005

Classen, A., Cordy, M., Schobbens, P.-Y., Heymans, P., Legay, A., & Raskin, J.-F. J.-F. (2013, aug). Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, *39*(8), 1069–1089.

Classen, A., et al. (2011). *Modelling and model checking variability-intensive systems* (Unpublished doctoral dissertation). Ph. D. thesis, PReCISE Research Center, Faculty of Computer Science, University of Namur, FUNDP.

Cohen, M., Dwyer, M., & Jiangfan Shi. (2008). Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, *34*(5), 633–650.

Czarnecki, K. (1998). *Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models* (Doctoral dissertation, Technical University of Ilmenau). Retrieved from `http://www.prakinf.tu-ilmenau.de/~czarn/diss/diss.pdf`

Czarnecki, K., Helsen, S., Con, S., Czarnecki, K., Eisenecker, U., Helsen, S., & Eisenecker, U. (2004). Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Concrete*, 1–28.

da Mota Silveira Neto, P. A., do Carmo Machado, I., McGregor, J. D., de Almeida, E. S., & de Lemos Meira, S. R. (2011). A systematic mapping study of software product lines testing. *Information and Software Technology*, *53*(5), 407–423.

Deelstra, S., Sinnema, M., & Bosch, J. (2004). Experiences in software product families: Problems and issues during product derivation. In R. L. Nord (Ed.), *Software product lines: Third international conference, splc 2004, boston, ma, usa, august 30-september 2, 2004. proceedings* (pp. 165–182). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-540-28630-1_10` doi: 10.1007/978-3-540-28630-1_10

Devroey, X., Perrouin, G., Cordy, M., Samih, H., Legay, A., Schobbens, P.-Y., & Heymans, P. (2015). Statistical prioritization for software product line testing: an experience report. *SoSyM*, 1–19.

Devroey, X., Perrouin, G., Legay, A., Cordy, M., Schobbens, P.-Y., & Heymans, P. (2014). Coverage Criteria for Behavioural Testing of Software Product Lines. In *Isola '14* (Vol. 8802, pp. 336–350). Springer.

Devroey, X., Perrouin, G., Legay, A., Schobbens, P.-Y., & Heymans, P. (2016). Search-based Similarity-driven Behavioural SPL Testing. In *Vamos '16* (pp. 89–96). ACM.

Dintzner, N., van Deursen, A., & Pinzger, M. (2016). FEVER: Extracting Feature-oriented Changes from Commits. In *Msr '16* (pp. 85–96). ACM. Retrieved from `http://dl.acm.org/citation.cfm?id=2901739.2901755http://dl.acm.org/citation.cfm?doid=2901739.2901755`

Engström, E., & Runeson, P. (2011). Software product line testing - A systematic mapping study. *Information and Software Technology*, *53*(1), 2–13.

Ensan, F., Bagheri, E., & Gašević, D. (2012). Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Caise '12* (pp. 613–628). Springer.

Gacek, C., & Anastasopoules, M. (2001). Implementing product line variabilities. *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, *26*(3), 109–117. Retrieved from `http://dl.acm.org/citation.cfm?id=375269` doi: 10.1145/379377.375269

Ganesan, D., Knodel, J., Kolb, R., Haury, U., & Meier, G. (2007). Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In *Software product line conference, 2007. splc 2007. 11th international* (pp. 74–83).

Garvin, B. J., Cohen, M. B., & Dwyer, M. B. (2011). Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, *16*(1), 61–102.

Gazzillo, P. (2015). *Kmax: Analyzing the linux build system* (Tech. Rep.). Department of Computer Science, New York University. Retrieved from `https://cs.nyu.edu/media/publications/TR2015-976.pdf`

Goguen, J. A. (1984, September). Parameterized programming. *IEEE Trans. Softw. Eng.*, *10*(5), 528–543. Retrieved from `http://dx.doi.org/10.1109/TSE.1984.5010277` doi: 10.1109/TSE.1984.5010277

Greiler, M., van Deursen, A., & Storey, M. A. (2012a). Test confessions: A study of testing practices for plug-in systems. In *Icse '12* (p. 244-254). ACM.

Greiler, M., van Deursen, A., & Storey, M. A. (2012b). Test confessions: A study of testing practices for plug-in systems. In *Icse '12* (p. 244-254). ACM.

Guo, J., Czarnecki, K., Apel, S., Siegmund, N., & Wasowski, A. (2013). Variability-aware performance prediction: A statistical learning approach. In *Ase '13* (pp. 301–311). IEEE.

Guo, J., White, J., Wang, G., Li, J., & Wang, Y. (2011). A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *Journal of Systems and Software*. Retrieved from `http://dx.doi.org/10.1016/j.jss.2011.06.026` doi: 10.1016/j.jss.2011.06.026

Hahsler, M., Grün, B., & Hornik, K. (2005, October). arules – A computational environment for mining association rules and frequent item sets. *Journal of Statistical Software*, *14*(15), 1–25. Retrieved from `http://dx.doi.org/10.18637/jss.v014.i15` doi: 10.18637/jss.v014.i15

Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., & Heymans, P. (2017).

Yo variability! jhipster: A playground for web-apps analyses. In *Proceedings of the eleventh international workshop on variability modelling of software-intensive systems* (pp. 44–51). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3023956.3023963 doi: 10.1145/3023956.3023963

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., & Le Traon, Y. (2014). Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering*, *40*(7), 650–670.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., & Traon, Y. L. (2013). Pledge: A product line editor and test generation tool. In *Proceedings of the 17th international software product line conference co-located workshops* (pp. 126–129). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2499777.2499778 doi: 10.1145/2499777.2499778

Hervieu, A., Baudry, B., & Gotlieb, A. (2011). PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *Issre '11* (pp. 120–129). IEEE.

Hetrick, W., Krueger, C., & Moore, J. (2006). Incremental return on incremental investment: Engenio's transition to software product line practice. *International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 798–804. Retrieved from http://dl.acm.org/citation.cfm?id=1176726 doi: 10.1145/1176617.1176726

Hubaux, a., Classen, a., & Heymans, P. (2009). Formal Modelling of Feature Configuration Workflow. *Proceedings of the 13th International Software Product Lines Conference (SPLC'09)*, 221–230. doi: 10.1109/RE.2009.55

IEEE Computer Society. (2014). *Guide to the software engineering body of knowledge (swebok(r)): Version 3.0* (3rd ed.; P. Bourque & R. E. Fairley, Eds.). Los Alamitos, CA, USA: IEEE Computer Society Press.

Jain, A., & Zongker, D. (1997). *Feature Selection: Evaluation, Application, and Small Sample Performance.*

JHipster. (2016). *JHipster.* Retrieved 2016-07-27, from http://jhipster.github.io/

Jin, D., Qu, X., Cohen, M. B., & Robinson, B. (2014). Configurations everywhere: implications for testing and debugging in practice. In *Icse '14 companion proceedings* (pp. 215–224). ACM.

Johansen, M. F. (2016). *Pairwiser.* Retrieved from https://inductive.no/pairwiser/

Johansen, M. F., Haugen, Ø., & Fleurey, F. (2012). An algorithm for generating t-wise covering arrays from large feature models. In *Splc '12* (Vol. 1, p. 46). ACM.

Johansen, M. F., Haugen, O., Fleurey, F., Eldegard, A. G., & Syversen, T. (2012). Generating better partial covering arrays by modeling weights on sub-product lines. In *Proceedings of the 15th international conference on model driven engineering languages and systems* (pp. 269–284). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dx.doi.org/10.1007/978-3-642-33666-9_18 doi: 10.1007/978-3-642-33666-9_18

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. (1990). *Feature-oriented domain analysis (foda) feasibility study* (Tech. Rep. No. CMU/SEI-90-TR-021). Pittsburgh,

PA: Software Engineering Institute, Carnegie Mellon University. Retrieved from `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231`

Kastner, C., & Apel, S. (2008). Type-checking software product lines-a formal approach. In *Ase '08* (pp. 258–267). IEEE.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., & Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Acm sigplan notices* (Vol. 46, pp. 805–824). ACM.

Kästner, C., Von Rhein, A., Erdweg, S., Pusch, J., Apel, S., Rendel, T., & Ostermann, K. (2012). Toward variability-aware testing. In *Fosd '12* (pp. 1–8). ACM.

Kniesel, G. (1999). Type-safe delegation for run-time component adaptation. In R. Guerraoui (Ed.), *Ecoop' 99 — object-oriented programming: 13th european conference lisbon, portugal, june 14–18, 1999 proceedings* (pp. 351–366). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/3-540-48743-3_16` doi: 10.1007/3-540-48743-3_16

Kyo, C. K., Lee, J., & Donohoe, P. (2002). Feature-Oriented Product Line Engineering. (August).

Lachmann, R., Lity, S., Lischke, S., Beddig, S., Schulze, S., & Schaefer, I. (2015). Delta-oriented test case prioritization for integration testing of software product lines. In *Splc '15* (pp. 81–90). ACM.

Lamancha, B. P., Polo, M., & Piattini, M. (2013). Systematic Review on Software Product Line Testing. In *Icsoft '10* (pp. 58–71). Springer.

Le Traon, Y. (2015, May). Combining multi-objective search and constraint solving for configuring large software product lines. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 1, p. 517-528). doi: 10.1109/ICSE.2015.69

Lochau, M., Oster, S., Goltz, U., & Schürr, A. (2012). Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, *20*(3-4), 567–604.

Lochau, M., Schaefer, I., Kamischke, J., & Lity, S. (2012). Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *Tap '12* (Vol. 7305, pp. 67–82). Springer.

Lopez-Herrejon, R. E., Chicano, F., Ferrer, J., Egyed, A., & Alba, E. (2013). Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *Icsme '13* (pp. 404–407). IEEE.

Lopez-Herrejon, R. E., Fischer, S., Ramler, R., & Egyed, A. (2015). A first systematic mapping study on combinatorial interaction testing for software product lines. In *Icstw '15* (pp. 1–10). IEEE.

Marijan, D., Gotlieb, A., Sen, S., & Hervieu, A. (2013). Practical pairwise testing for software product lines. In *Splc '13* (p. 227). ACM.

Mathur, A. P. (2008). *Foundations of software testing.* India: Pearson Education.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., & Apel, S. (2016, 5). A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th international conference on software engineering (icse).* New York, NY: ACM Press.

Meinicke, J., Wong, C.-P., Kästner, C., Thüm, T., & Saake, G. (2016, 9). On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st ieee/acm international conference on automated software engineering (ase)*. New York, NY: ACM Press.

Nebut, C., Traon, Y. L., & Jézéquel, J. (2006). System testing of product lines: From requirements to test cases. In *Software product lines* (pp. 447–477). Springer.

Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., & Kulesza, U. (2015). Safe evolution templates for software product lines. *Journal of Systems and Software*, *106*, 42–58.

Nguyen, H. V., Kästner, C., & Nguyen, T. N. (2014a). Exploring variability-aware execution for testing plugin-based web applications. In *Icse '14* (pp. 907–918).

Nguyen, H. V., Kästner, C., & Nguyen, T. N. (2014b). Exploring variability-aware execution for testing plugin-based web applications. In *Icse '14* (pp. 907–918).

Ochoa, L., Pereira, J. A., González-Rojas, O., Castro, H., & Saake, G. (2017). A survey on scalability and performance concerns in extended product lines configuration. In *Proceedings of the eleventh international workshop on variability modelling of software-intensive systems* (pp. 5–12). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/3023956.3023959` doi: 10.1145/3023956.3023959

Osmani, A., Sorhus, S., Hartig, P., Sawchuk, S., Boudrias, S., Ford, B., ... Verschaeve, A. (2012). *Yeoman Website.* Retrieved 2016-07-27, from `http://yeoman.io/`

Oster, S., Zorcic, I., Markert, F., & Lochau, M. (2011a). Moso-polite: Tool support for pairwise and model-based software product line testing. In *Proceedings of the 5th workshop on variability modeling of software-intensive systems* (pp. 79–82). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1944892.1944901` doi: 10.1145/1944892.1944901

Oster, S., Zorcic, I., Markert, F., & Lochau, M. (2011b). Moso-polite: Tool support for pairwise and model-based software product line testing. In *Vamos '11* (pp. 79–82). ACM.

Parejo, J. A., Sánchez, A. B., Segura, S., Ruiz-Cortés, A., Lopez-Herrejon, R. E., & Egyed, A. (2016). Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, *122*, 287–310.

Parnas, D. L. (1976). On the design and development of program families. *IEEE Transactions on software engineering*(1), 1–9.

Pérez Lamancha, B., & Polo Usaola, M. (2010). Testing Product Generation in Software Product Lines Using Pairwise for Features Coverage. In A. Petrenko, A. Simão, & J. C. Maldonado (Eds.), *Ictss '10* (pp. 111–125). Springer.

Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., & le Traon, Y. (2011). Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, *20*(3-4), 605–643.

Perrouin, G., Sen, S., Klein, J., Baudry, B., & Traon, Y. l. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 third international conference on software testing, verification and validation* (pp. 459–468). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/ICST.2010.43` doi: 10.1109/ICST.2010.43

Pohl, K., Böckle, G., & Linden, F. J. v. d. (2005). *Software product line engineering: Foundations, principles and techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Pohl, K., Metzger, A., & Pohl, K. (2006). Variability Management in Software Product Line Engineering. *29th International Conference on Software Engineering {(ICSE} 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume*, 186–187. Retrieved from `http://doi.ieeecomputersociety.org/10.1109/ICSECOMPANION.2007.83{%}5Cnhttp://doi.acm.org/10.1145/1134499` doi: 10.1109/ICSECOMPANION.2007.83

Qu, X., Cohen, M. B., & Rothermel, G. (2008). Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on software testing and analysis* (pp. 75–86).

Rothermel, G., Harrold, M. J., Ostrin, J., & Hong, C. (1998). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the international conference on software maintenance* (pp. 34–). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dl.acm.org/citation.cfm?id=850947.853294`

Rothermel, G., Harrold, M. J., von Ronne, J., & Hong, C. (2002). Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, *12*(4), 219–249. Retrieved from `http://dx.doi.org/10.1002/stvr.256` doi: 10.1002/stvr.256

Sampaio, G., Borba, P., & Teixeira, L. (2016). Partially safe evolution of software product lines. In *Splc '16* (pp. 124–133). ACM.

Sánchez, A. B., Segura, S., Parejo, J. A., & Ruiz-Cortés, A. (2015). Variability testing in the wild: the Drupal case study. *Software and Systems Modeling*. doi: 10.1007/s10270-015-0459-z

Sánchez, A. B., Segura, S., Parejo, J. A., & Ruiz-Cortés, A. (2017). Variability testing in the wild: the drupal case study. *Software & Systems Modeling*, *16*(1), 173–194. Retrieved from `http://dx.doi.org/10.1007/s10270-015-0459-z` doi: 10.1007/s10270-015-0459-z

Sánchez, A. B., Segura, S., & Ruiz-Cortés, A. (2013a). The drupal framework: A case study to evaluate variability testing techniques. In *Proceedings of the eighth international workshop on variability modelling of software-intensive systems* (pp. 11:1–11:8). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2556624.2556638` doi: 10.1145/2556624.2556638

Sánchez, A. B., Segura, S., & Ruiz-Cortés, A. (2013b). The drupal framework: A case study to evaluate variability testing techniques. In *Proceedings of the eighth international workshop on variability modelling of software-intensive systems* (pp. 11:1–11:8). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/2556624.2556638` doi: 10.1145/2556624.2556638

Sanchez, A. B., Segura, S., & Ruiz-Cortes, A. (2014). A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *Icst '14* (pp. 41–50). IEEE.

Sanchez, S. (2012). Automated testing on the analysis of variability-intensive artifacts : An exploratory study with sat solvers.

Sarkar, A., Guo, J., Siegmund, N., Apel, S., & Czarnecki, K. (2015). Cost-Efficient

Sampling for Performance Prediction of Configurable Systems (T). In *Ase '15* (pp. 342–352). IEEE.

Sayyad, A. S., Menzies, T., & Ammar, H. (2013). On the value of user preferences in search-based software engineering: A case study in software product lines. In *Icse '13* (pp. 492–501). IEEE.

Schmid, K., & John, I. (2004). A customizable approach to full lifecycle variability management. *Science of Computer Programming*, *53*(3), 259–284. doi: 10.1016/j.scico.2003.04.002

She, S., Lotufo, R., Berger, T., Wasowski, A., & Czarnecki, K. (2010, January). The variability model of the linux kernel. In *Fourth international workshop on variability modelling of software-intensive systems (vamos'10).* Linz, Austria.

Siegmund, N., Grebhahn, A., Apel, S., & Kästner, C. (2015). Performance-influence models for highly configurable systems. In *Esec/fse '15* (pp. 284–294). ACM.

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., & Saake, G. (2012). Predicting performance via automated feature-interaction detection. In *Icse '12* (pp. 167–177). IEEE.

Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P. G., Apel, S., & Kolesnikov, S. S. (2013). Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, *55*(3), 491–507.

Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., & Saake, G. (2008). Measuring Non-functional properties in software product lines for product derivation. *Neonatal, Paediatric and Child Health Nursing*, 187–194. doi: 10.1109/APSEC.2008.45

Šmeral, R. (2014). Modern Performance Tools Applied.

Solís, C., & Wang, X. (2011). A study of the characteristics of behaviour driven development. *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011*, 383–387. doi: 10.1109/SEAA.2011.76

ter Beek, M. H., Fantechi, A., Gnesi, S., & Mazzanti, F. (2015, nov). Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*.

Thüm, T., Apel, S., Kästner, C., Kuhlemann, M., Schaefer, I., & Saake, G. (2012). Analysis Strategies for Software Product Lines. , 35. Retrieved from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.366.3306{&}rep=rep1{&}type=pdf`

Thüm, T., Apel, S., Kästner, C., Schaefer, I., & Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, *47*(1), 6:1–6:45. Retrieved from `http://doi.acm.org/10.1145/2580950` doi: 10.1145/2580950

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014, January). Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, *79*, 70–85. Retrieved from `http://dx.doi.org/10.1016/j.scico.2012.06.002` doi: 10.1016/j.scico.2012.06.002

Valov, P., Guo, J., & Czarnecki, K. (2015). Empirical comparison of regression methods for variability-aware performance prediction. In *Splc '15* (pp. 186–190). ACM.

Van Deursen, A., & Klint, P. (2002). Domain-Specific Language Design Requires Feature Descriptions . , 1–17.

Van Gurp, J., Bosch, J., & Svahnberg, M. (2001). On the notion of variability in software product lines. In *Proceedings of the working ieee/ifip conference on software architecture* (pp. 45–). Washington, DC, USA: IEEE Computer Society. Retrieved from `http://dx.doi.org/10.1109/WICSA.2001.948406` doi: 10.1109/WICSA.2001.948406

Von Rhein, A., Apel, S., Kästner, C., Thüm, T., & Schaefer, I. (2013). The pla model: on the combination of product-line analyses. In *Vamos '13* (p. 14). ACM.

Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., & Bodden, E. (2014). Variational data structures: Exploring tradeoffs in computing with variability. In *Splash '14* (pp. 213–226). ACM.

Wong, W. E., Horgan, J. R., London, S., & Mathur, A. P. (1998, April). Effect of test set minimization on fault detection effectiveness. *Softw. Pract. Exper.*, *28*(4), 347–369. Retrieved from `http://dx.doi.org/10.1002/(SICI)1097-024X(19980410)28:4<347::AID-SPE145>3.0.CO;2-L` doi: 10.1002/(SICI)1097-024X(19980410)28:4⟨347::AID-SPE145⟩3.0.CO;2-L

Yilmaz, C., Cohen, M. B., & Porter, A. A. (2006). Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, *32*(1), 20–34.

Yoo, S., & Harman, M. (2007). Regression Testing Minimisation, Selection and Prioritisation : A Survey. *Test. Verif. Reliab*, *00*, 1–7. doi: 10.1002/000

Zave, P. (1993, Aug). Feature interactions and formal specifications in telecommunications. *Computer*, *26*(8), 20-28. doi: 10.1109/2.223539

Zhang, Y., Guo, J., Blais, E., Czarnecki, K., & Yu, H. (2016). A mathematical model of performance-relevant feature interactions. In *Splc '16* (pp. 25–34). ACM.

A. List of JHipster questions

We present in this appendix the different questions prompted to the user during the configuration phase of JHipster 3.6.1. The number in front of each question doesn't represent its place in all configuration process. If it is true to say that question 1 will always be asked before question 2, question 3 won't always be prompted (if the answer to question 1 is *'Monolithic Application'*, for instance). This list can also be found on JHipster website: `https://jhipster.github.io/creating-an-app/#2`.

We also present the different constraints we identified between the different technologies. A description of these frameworks can be found in Appendix B.

These questions along, with the constraints, are the implementation of JHipster variability and were extracted from the JavaScript files (as presented in Chapter 4.3).

Questions	**Possible Answers**
1) Which *type* of application would you like to create?	Monolithic Microservice Application Microservice Gateway Uaa Server
2) What is the base name of your application?	String input
3) As you are running in a microservice architecture, on which port would you like your server to run?	Integer
4) What is your default Java package name?	String input
5) Which *type* of authentication would you like to use?	HTTP Session Authentication OAuth2 Authentication JWT Authentication UAA Authentication
6) What is the folder path of your UAA application?	String input
7) Do you want to use social login (Google, Facebook, Twitter)?	No Yes
8) Which *type* of database would you like to use?	SQL MongoDB Cassandra No database
9) Which *production* database would you like to use?	MySQL PostgreSQL Oracle MariaDB
10) Which *development* database would you like to use?	H2 disk-based persistence H2 in-memory persistence MySQL PostgreSQL Oracle 12c MariaDB
11) Do you want to use Hibernate 2nd level cache?	No Yes, with ehcache Yes, with HazelCast
12) Do you ant to use a search engine in your application?	No Yes, ElasticSearch
13) Do you want to use clustered HTTP sessions?	No Yes, HazelCast
14) Do you want to use WebSockets?	No Yes, with Spring Websocket
15) Would you like to use Maven or Gradle for building the backend?	Maven Gradle
16) Would you like to use the LibSass stylesheet pre-processor for your css?	Yes No
17) Would you like to enable internationalization support? - Please choose the native language of the application - Please choose additional languages to install	Yes No
18) Which testing frameworks would you like to use?	Gatling Cucumber Protractor No

Table A.1.: List of questions and answers for *yo jhipster* command

B. Technologies description

We briefly present in this Appendix the different technologies/frameworks JHipster offers. We aim at summarizing these technologies rather than giving a complete overview of all possibilities they offer. All information presented here can be found on JHipster official website[1] and on the respective websites of the frameworks.

B.1. Application Type

We present hereafter the different *application types* one can produce with JHipster. These 4 types of application are the ones available at question 1. Besides these 4 types, we can also generate client or server standalones using JHipster sub-generators. Microservice applications, gateways and UAA servers are all parts of the microservice architecture which separates client and server parts to ease the scaling.

Application Type	**Description**
Microservice Application	Microservices are applications managing REST requests. They are stateless and several instances of these can be run in parallel to handle heavy traffic.
Microservice Gateway	A *gateway* is an application that manages web traffic and serves an AngularJS application. There may be several different gateways, to follow Backends for Frontends pattern for example.
Monolithic Application	A Monolithic Application is a classical web application relying on Spring Boot on the server side and AngularJS on the client side. The JHipster Team recommend this option if there aren't any specific requirements.
UAA Server	An User Account and Authentication (UAA) server is used in microservice architectures to offer OAuth2 authentication to the gateway.

Table B.1.: Application types description

[1] `https://jhipster.github.io/creating-an-app`

B.2. Authentication Type

JHipster offers 4 different kind of authentication. HTTP Session and OAuth2 available in Monolithic Applications; JSON Web Token (JWT) for both Monolithic and Microservice applications; and UAA available only in Microservice architectures.

Authentication Type	Description
HTTP Session	Classic statefull authentication mechanism based on HTTP sessions.
OAuth2	Stateless security mechanism using a secret key. JHipster generate an Authentication Server, based on Spring Security, which delivers tokens.
JWT	Stateless security mechanism similar to OAuth2. It doesn't require a persistence mechanism and can then work on all SQL and NoSQL databases. It uses a signed secure token holding the user's login name and authorities.
UAA Server	OAuth2 authentication for microservices.

Table B.2.: Authentication types description

B.3. Databases

JHipster supports 8 different databases: six SQL and two NOSQL. Among these databases, 2 (H2 disk-based and H2 in-memory) are only available as development databases. JHipster also offers the possibility to create microservices without any database. More information on the different databases can be found on their respective websites.
SQL databases

- MySQL
- PostgreSQL
- MariaDB
- H2 (in-memory and disk-based)
- Oracle

NoSQL databases

- MongoDB
- Cassandra

B.4. Testing frameworks

All JHipster variants come with Java unit tests (Spring Test Context framework) and JavaScript unit tests (using Karma.js). Besides these tests, JHipster offers 3 additional frameworks to its users, each of which focuses on a specific aspect of the web application. These frameworks can't, however, be selected in all configurations. See Appendix A for more information.

Testing Framework	Description
Cucumber	Cucumber is a framework for behaviour-driven development. It is used to assert the validity of certain scenarios, among which, for instance, CRUD operations on the database.
Gatling	Gatling is a load testing framework used for performance testing.
Protractor	Protractor is an UI integration testing framework. It simulates users interaction with the application by launching a web browser and interacting with the interface (for instance, it tries to log in by clicking the log-in button and entering credentials).

Table B.3.: Testing frameworks description

B.5. Others

We present in this section frameworks which didn't find a place in previous section.

Framework	Description
Social Login	Offers the possibility to log into a JHipster application using major social networks (Google, Facebook and Twitter) accounts.
ElasticSearch	RESTful distributed search engine adding search capabilities on top of the database. It is currently only available with SQL databases.
EhCache	Local Hibernate (Java Persistence API provider) cache aiming at improving performance and simplifying scaling.
Hazelcast	Used as Hibernate cache, it is similar to EhCache although it is a distributed cache more adapted to clustered environment. It can also be used for clustered HTTP sessions which allow to replicate sessions in a cluster to prevent loss of sessions (for applications using web sessions) in case the server goes down.
WebSockets	WebSockets specification defines an API that allows web pages to use the WebSocket protocol for two-way communication with a remote host. It presents the WebSocket interface and defines a full-duplex communication channel that operates through a single socket over the Web. WebSockets offer a huge reduction of the traffic network and the latency.
LibSass	Style-sheet pre-processor to simplify the design of CSS and to treat conditional styling (for instance, if the condition is true use this font, else use this one).
Maven	Apache Maven is a software project management and comprehension tool, based on the concept of Project Object Model (POM). It is one of the two options to handle the complete building process of a JHipster app (dependencies management, building process, etc.).
Gradle	Similar to Maven, Gradle is a modern open source polyglot build automation system. Whereas Maven is seen as more stable and mature, Gradle is deemed more flexible and easier to extend.

Table B.4.: Other frameworks description

C. Entities JDL scripts

C.1. Base model

As we presented in Section 4.4.2, the base model we used to generate entities is the one the JHipster team present as an example:

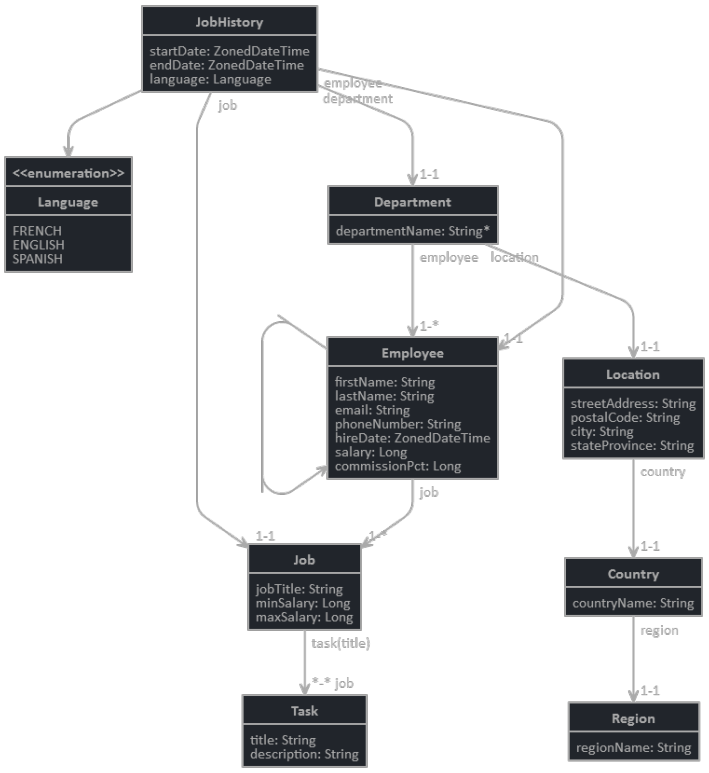

Figure C.1.: JHipster JDL entities example

C.2. MySQL and PostgreSQL databases JDL

The JDL script for MySQL and PostgreSQL databases, both SQL databases, is directly derived from Figure C.1. It is also available on JHipster Domain Language Studio web

page: `https://jhipster.github.io/jdl-studio/`. For conciseness sake, we removed comments from the script below.

Listing C.1: SQL databases JDL script

```
entity Region { regionName String}

entity Country { countryName String}

entity Location {
        streetAddress String,
        postalCode String,
        city String,
        stateProvince String
}

entity Department { departmentName String required}

entity Task {
        title String,
        description String
}

entity Employee {
        firstName String,
        lastName String,
        email String,
        phoneNumber String,
        hireDate ZonedDateTime,
        salary Long,
        commissionPct Long
}

entity Job {
        jobTitle String,
        minSalary Long,
        maxSalary Long
}

entity JobHistory {
        startDate ZonedDateTime,
        endDate ZonedDateTime,
        language Language
}

enum Language { FRENCH, ENGLISH, SPANISH}

relationship OneToOne { Country{region} to Region}
relationship OneToOne {Location{country} to Country}
relationship OneToOne {Department{location} to Location}
relationship ManyToMany {Job{task(title)} to Task{job}}
relationship OneToMany {
        Employee{job} to Job,
        Department{employee} to Employee
}
relationship ManyToOne {Employee{manager} to Employee}
relationship OneToOne {
        JobHistory{job} to Job,
        JobHistory{department} to Department,
        JobHistory{employee} to Employee
}

paginate JobHistory, Employee with infinite-scroll
paginate Job with pagination
dto * with mapstruct
service all with serviceImpl except Employee, Job
angularSuffix * with mySuffix
```

C.3. MongoDB database JDL

MongoDB being a NoSQL database, relationships are not allowed in the JDL script. We adapted the previous script by simply removing those relationships declaration. The

remainder of the code is exactly the same, as presented in Listing C.2.

Listing C.2: MongoDB JDL script

```
entity Region { regionName String}

entity Country { countryName String}

entity Location {
        streetAddress String,
        postalCode String,
        city String,
        stateProvince String
}

entity Department { departmentName String required}

entity Task {
        title String,
        description String
}

entity Employee {
        firstName String,
        lastName String,
        email String,
        phoneNumber String,
        hireDate ZonedDateTime,
        salary Long,
        commissionPct Long
}

entity Job {
        jobTitle String,
        minSalary Long,
        maxSalary Long
}

entity JobHistory {
        startDate ZonedDateTime,
        endDate ZonedDateTime,
        language Language
}

enum Language { FRENCH, ENGLISH, SPANISH}

paginate JobHistory, Employee with infinite-scroll
paginate Job with pagination
dto * with mapstruct
service all with serviceImpl except Employee, Job
angularSuffix * with mySuffix
```

C.4. Cassandra database JDL

Cassandra posed more constraints regarding the JDL. Indeed, being a NoSQL database like MongoDB, we first needed to remove the relationships declaration. Beside this common constraints, we also had trouble with the type *ZonedDateTime* which is not supported with Cassandra. We also had an issue with the enumeration *Language*, which we transformed in a *String* attribute. Finally, we also had to remove the *paginate* options, resulting in Listing C.3.

Listing C.3: Cassandra JDL script

```
entity Region { regionName String}

entity Country { countryName String}

entity Location {
        streetAddress String,
        postalCode String,
        city String,
        stateProvince String
}

entity Department { departmentName String required}

entity Task {
        title String,
        description String
}

entity Employee {
        firstName String,
        lastName String,
        email String,
        phoneNumber String,
        hireDate Date,
        salary Long,
        commissionPct Long
}

entity Job {
        jobTitle String,
        minSalary Long,
        maxSalary Long
}

entity JobHistory {
        startDate Date,
        endDate Date,
        language String
}

dto * with mapstruct
service all with serviceImpl except Employee, Job
angularSuffix * with mySuffix
```

D. Possibilities on the JHipster data-set

As introduced in the Section 4.4.2, we decide to use a CSV file to store all data concerning functional analyses. The execution of the testing workflow yielded a large file comprising numerous results for each configuration. This file available at `https://github.com/axel-halin/Thesis-JHipster/blob/master/Results/jhipster.csv` allows to identify failing configurations, i.e., configurations that do not generate, compile or build. An excerpt of this file is presented in Table D.1. In addition of these results, we also extracted and exploited stack traces for each variants (when it doesn't generate/-compile/build) allowing us to find the cause of the failure. This section presents, first, all the types of the data-set which is stored in a CSV file format. Then, we provide with the data-set presented in Section D.2 a ground truth for various testing techniques and methodologies. We will present some possibilities that can be achieved with this data-set using R[1] which is a free software environment for statistical computing and graphics. We will present successively the use of association rules and the creation of each figures presented in this master thesis. We also provide R scripts to illustrate the possibilities and to give the opportunity to use them to replicate the study.

JHipster Register	**Docker**	**application Type**	**authentication Type**	**(...)**	**Compile**	**Log-Compile**	**Build**	**Log-Build**
jhipster1	true	"monolith"	"session"	(...)	OK		KO	Failed to get driver instance for jdbcUrl=jdbc:mariadb://mariadb:3306/jhipster
jhipster1	false	"monolith"	"session"	(...)	OK		KO	Failed to get driver instance for jdbcUrl=jdbc:mariadb://localhost:3306/jhipster
(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)	(...)
jhipster13128	true	"monolith"	"session"	(...)	OK		OK	ND
jhipster13128	false	"monolith"	"session"	(...)	OK		OK	ND

Table D.1.: CSV file excerpt

D.1. Data-set content

Table D.2 describes headers and entities from the data-set. All descriptions of the features are explained in Appendix B. Some cells from the CSV file are "ND" values ("ND" is

[1] `https://www.r-project.org/`

an acronym for "Not Defined"). These values may happen in some configurations. For instance, when a developer choose to work on a *micro-service* application then features *clusteredHttpSession*, *websocket*, *enableSocialSignIn* and *useSass* will imply a "ND" value because questions about these technologies will be not prompted to the developer (See details on the configuration process in Section 3.2.1). Non-functional analyses are not available for the moment but it's in order to have it in future works.

Headers	**Entities**
JHipsterRegister	Number of the folder where the web-app is generated/compiled/build and tested.
Docker	Boolean value which is *true* when Docker is used to deploy the web-app, *false* otherwise
applicationType	*monolith*,*gateway*,*microservice* and *uaa* are the possible type of application provided by JHipster.
authenticationType	*session*,*jwt*,*uaa*,*oauth2* are the possibilities for the type of authentication
hibernateCache	*hazelcast*,*ehcache* are the type of hibernateCache. The developer has the possibility to not have hibernateCache (*no* value).
clusteredHttpSession	*hazelcast* is the value for clusteredHttpSession if the developer want it, otherwise it's a *no* value.
websocket	*spring-websocket* is the value for websocket if the developer want it, otherwise it's a *no* value.
databaseType	SQL database (*sql*) or NoSQL databases (*mongodb* and *cassandra*)
devDatabaseType	SQL databases (*mysql*, *mariadb* or *postgresql*) or NoSQL databases (*mongodb* or *cassandra*) or H2 databases (*DiskBased*, *inMemory*).
prodDatabaseType	The same from devDatabaseType but no H2 database.
buildTool	*maven* and *gradle* are the two build tool provided by JHipster.
searchEngine	*elasticsearch* or *no* if the developer doesn't want a search engine in his web-app.
enableSocialSignIn	Boolean value which is true when the developer want a Social SignIn in his web-app, false otherwise.
useSass	Boolean value which is true when the developer uses useSass in his web-app, *false* otherwise.
enableTranslation	Boolean value which is true when the developer uses any translations in his web-app, *false* otherwise.
testFrameworks	List of activated framework testing : *cucumber*, *protractor* and *gatling*.
Generate	*OK* value if the generation succeed else *KO*.
Log-Generate	Logs that we extract if the web-app fail to generate.
Compile	*OK* value if the compilation succeed else *KO*.
Log-Compile	Logs that we extract if the web-app fail to compile.
Build	*OK* value if the build succeed else *KO*.
Log-Build	Logs that we extract if the app fail to deploy.

Table D.2.: jhipster.csv file description

D.2. Association Rule learning method

As explained in Section 6.2, we used Association Rule learning method (Hahsler et al., 2005) in the focus to investigate correlations between features and the failure results. This method aims at extracting relations between variables of large data-sets and outputs a set of rules.

To extract the rules presented in Table 6.1 from the method, we used a R package: *arules*[2] which provides the infrastructure for representing, manipulating and analysing transaction data and patterns (frequent itemsets and association rules).

We focused ourselves to rules where the RHS was *Compile=KO* and because there are no failure at the generation phase. Listing D.1 gives us rules for the compilation phase. Then, we were interested in rules where the RHS was *Build=KO*. Listing D.2 presents the code we made.

See Section 6.2 for more details on the choices we did (for instance, length of the rules we fixed, etc).

Listing D.1: Arules Compilation R Script

```
data<-read.csv(file="jhipster.csv", na.strings = c("", "NA"), head=TRUE, sep=',')

library(arules)

subData <- data.frame(data$Docker,data$applicationType,data$authenticationType,
data$hibernateCache,data$clusteredHttpSession,data$websocket,data$devDatabaseType,
data$prodDatabaseType,data$buildTool,data$searchEngine,data$enableSocialSignIn,
data$useSass,data$Build)

rules <- apriori(subData, parameter = list(minlen=2,
    maxlen=4,confidence=1,support=0.0001, target =
    'rules'),appearance=list(rhs=c('data.Compile=OK'),default='lhs'))
rules <- sort(rules, by = 'support')

## redundant rules non redondunt
inspect(rules[!is.redundant(rules)])
```

Listing D.2: Arules Build R Script

```
data<-read.csv(file="results3.csv", na.strings = c("", "NA"), head=TRUE, sep=',')

library(arules)

subData <- data.frame(data$Docker,data$applicationType,data$authenticationType,
data$hibernateCache,data$clusteredHttpSession,data$websocket,data$devDatabaseType,
data$prodDatabaseType,data$buildTool,data$searchEngine,data$enableSocialSignIn,
data$useSass,data$Build)

rules <- apriori(subData, parameter = list(minlen=2,
    maxlen=4,confidence=1,support=0.0001, target =
    'rules'),appearance=list(rhs=c('data.Build=OK'),default='lhs'))
rules <- sort(rules, by = 'support')

## redundant rules non redondunt
inspect(rules[!is.redundant(rules)])
```

[2] https://cran.r-project.org/web/packages/arules/index.html

Listing D.3: Proportion of build failure by feature: figure R script

```
data<-read.csv(file="jhipster.csv", na.strings = c("", "NA"), head=TRUE, sep=',')

data1 <- data.frame(table(data$Build,data$applicationType))
data2 <- data.frame(table(data$Build,data$authenticationType))

library(plyr)
library(scales)

names(data1)[names(data1)=="Var1"] <- "Build"
names(data1)[names(data1)=="Var2"] <- "Feature"
names(data2)[names(data2)=="Var1"] <- "Build"
names(data2)[names(data2)=="Var2"] <- "Feature"

data1 <- ddply(data1, "Feature", transform,Percentage = Freq / sum(Freq) * 100)
data2 <- ddply(data2, "Feature", transform,Percentage = Freq / sum(Freq) * 100)

data1$Feature <-
    as.data.frame(sapply(data1$Feature,gsub,pattern="uaa",replacement="uaaApp"))
data1$Feature <- unlist(data1$Feature)

data2$Feature <-
    as.data.frame(sapply(data2$Feature,gsub,pattern="uaa",replacement="uaaAuth"))
data2$Feature <- unlist(data2$Feature)

library(plyr)
dataAll <- rbind.fill(data1, data2)
print(dataAll)

library(ggplot2)

ggplot(dataAll, aes(x=Feature, y=Percentage, fill=Build,
    order=desc(Feature)),xlab='') +
geom_bar(stat="identity",colour="black") +
theme(axis.title.x=element_blank()) +
guides(fill=guide_legend(reverse=TRUE)) +
geom_text(aes(label = percent(Percentage/ 100), x =
    Feature),position=position_stack(vjust = 0.5), size = 2,colour = "black")

ggsave("bugsFeatures.pdf",height = 7, width = 9)
```

D.3. Figures

We present here the code created for the different plots of our thesis. Respectively, Listing D.3 and Listing D.4 are the codes we achieved to create Figure 6.1 and Figure 6.2.

D.3.1. Faults and failures detection efficiency of sampling techniques

We present in this Section the script used to generate Figure 6.3 and Figure 6.4 (Listing D.5).

To generate this Figure, we first ran simulations on random sets of size 1 to 2500. We simulated 100 random selections to determine the point where all faults were discovered.

Listing D.4: Proportion of failures by fault: figure R script

```
data<-read.csv(file="results3.csv", na.strings = c("", "NA"), head=TRUE, sep=',')

library(plyr)
library(scales)

data <- ddply(data, "Bugs", transform,Percentage = percent(((Failures / 9376) *
    100) / 100))

library(ggplot2)

data <- data[order(data$Failures, decreasing = TRUE),]

print(data)

ggplot(data,aes(reorder(Bugs, Failures), Failures),aes(x=Bugs,
    y=Failures),xlab='',label = Percentage) +
geom_bar(stat="identity",fill = "#56B4E9") +
theme(
axis.title.y=element_blank()) +
geom_text(aes(label = Percentage, y = Failures/2), size = 3,colour = "black") +
    coord_flip()

ggsave("barplot.pdf",width=8,height = 4.5)
```

We computed the percentage of failures on the basis of the total failure percentage ($\approx$ 34.37%). We then generated the plot with the library *ggplot*.

Listing D.5: Faults and Failures found by sampling techniques - R script

```
library(ggplot2)
library(plyr)

file <- "/home/axel/samplingComparison3.csv"

df <- read.csv2(file, header = TRUE, sep = ";", quote = "\"", dec = ".", fill = TRUE)
df$Sampling.technique <- "Random"

for(sampleSize in 401:2500){
de <- data.frame("Random", sampleSize, floor((35.71*sampleSize)/100), floor((35.71*sampleSize)/100))
names(de) <- c("Sampling.technique","Sample.size","Failures","Failures.mean")
df <- rbind.fill(df,de)
}

df["Failures.mean"] <- NA
for(sampleSize in 1:2500){df[df$Sample.size==sampleSize,]$Failures.mean <- summary(df[df$Sample.size==sampleSize,]$Failures)[4]}

Sampling.technique <- c("1-wise", "2-wise", "3-wise", "4-wise", "Most-enabled-disabled", "One-disabled", "One-enabled","PLEDGE(8)","PLEDGE(12)", "PLEDGE(41)","PLEDGE(126)","PLEDGE(374)")
Sample.size <- c(8, 41, 126, 374, 574, 922, 2340,8,12,41,126,374)
Bugs.mean <- c(2, 5, 6, 6, 2, 5, 6,2.14,2.82,4.70,4.66,4.62)
Failures.mean <- c(2, 14, 52, 161, 190, 253, 872,3.16,4.92,17.64,49.08,139.20)

df <- rbind.fill(df, data.frame(Sampling.technique, Sample.size, Bugs.mean, Failures.mean))


ggplot(df, aes(x=Sample.size, y=Bugs.mean, shape=Sampling.technique, color=Sampling.technique)) +
  geom_point(size=4) +
  scale_shape_manual(values=c(15,16,17,18,19,20,21,9,10,11,12,13,46))+
  geom_smooth(method=loess,se=FALSE) +
  xlab("Configurations") + ylab("Faults found") + labs(shape="Sampling", colour="Sampling")

ggplot(df, aes(x=Sample.size, y=Failures.mean, shape=Sampling.technique, color=Sampling.technique)) +
  geom_point(size=4) +
  scale_shape_manual(values=c(15,16,17,18,19,20,21,9,10,11,12,13,46))+
  geom_smooth(method=loess,se=FALSE) +
  xlab("Configurations") + ylab("Failures found") + labs(shape="Sampling", colour="Sampling")
```

E. Grid'5000 resources overview

To run our experiments on Grid'5000, we mainly relied on one site: Rennes. We ran several small scale experiments in Lille's site, when Rennes was unavailable (due to lack of available machines or maintenance).

With transparency in mind, we present here the different machine used to run, and distribute the execution of, the workflow. More details can be found on Grid'5000 official website: `https://www.grid5000.fr/mediawiki/index.php/Rennes:Home`.

Rennes offers 173 different machines scattered across 5 clusters:

Parapluie Cluster

- **Introduction date:** *2010*
- **Number of nodes:** *40*
- **CPUs:** *(2x) AMD Opteron 6164 HE – 12 cores/CPU*
- **Memory:** *48GB RAM*
- **Disk space:** *232GB HDD*
- **10Gbps ethernet:** *no*

Parapide Cluster

- **Introduction date:** *2010*
- **Number of nodes:** *25*
- **CPUs:** *(2x) Intel Xeon X5570 – 4 cores/CPU*
- **Memory:** *24GB RAM*
- **Disk space:** *465GB HDD*
- **10Gbps ethernet:** *no*

Paranoia Cluster

- **Introduction date:** *2014*
- **Number of nodes:** *8*
- **CPUs:** *(2x) Intel Xeon E5-2660 v2 – 10 cores/CPU*
- **Memory:** *128GB RAM*
- **Disk space:** *(5x) 558GB HDD*
- **10Gbps ethernet:** *yes*

Parsilo Cluster

- **Introduction date:** *2015*
- **Number of nodes:** *28*
- **CPUs:** *(2x) Intel Xeon E5-2630 v3 – 8 cores/CPU*
- **Memory:** *128GB RAM*
- **Disk space:** *(5x) 558GB HDD + 186GB SSD*
- **10Gbps ethernet:** *yes*

Paravance Cluster

- **Introduction date:** *2015*
- **Number of nodes:** *72*
- **CPUs:** *(2x) Intel Xeon E5-2630 v3 – 8 cores/CPU*
- **Memory:** *128GB RAM*
- **Disk space:** *(2x) 558GB HDD*
- **10Gbps ethernet:** *yes*

F. Faults listing

During the course of this study, and as one of the objectives, we identified and classified the different faults we encountered. We present them in this Appendix.

The faults we found mostly occurred because of a combination of features related to the authentication type or the database type. These faults being due to a combination of features, we choose to present them in the same order as in Section 6.3.

F.1. MariaDB with Docker

This first fault is related to the use of *MariaDB* in the variant. This fault appears when trying to build the variant with Docker, regardless of the build tool used in the configuration.

It is however not found in all MariaDB variants. Indeed, the root cause of this issue is a missing line in the file *src/main/docker/app.yml* (see Listing F.1 and Listing F.2). This missing line causes Docker to look for a wrong repository/tag, one that doesn't exist. This fault manifests itself with a error message in the build log: *Error parsing reference: "jhipster - jhipster-mariadb" is not a valid repository/tag* or *Error parsing reference: "jhipster - jhipster-mariadb:mariadb - jhipster-registry" is not a valid repository/tag.*

When inspecting the template, we can see that the line "*external_links:*" is only activated if:

- The production database is MySQL, PostgreSQL, MongoDB or Cassandra;
- The application is a Microservice Application or Gateway;
- ElasticSearch is enabled

This fault was fixed[1] on November 24th 2016.

Listing F.1: Erroneous app.yml

```
1 image: jhipster
2       - jhipster-mariadb:mariadb
```

Listing F.2: Valid app.yml

```
1 image: jhipster
2 external_links:
3       - jhipster-mariadb:mariadb
```

[1] https://github.com/jhipster/generator-jhipster/commit/9f320bc86ab6209585e24b4db8abecd3ecf29ba2

F.2. MariaDB using Gradle

This issue concerns variants relying on Gradle as the build tool and MariaDB as the database. It is due to a missing dependency in the generator Gradle template. When attempting to build such a variant, the console will prompt one of the following exceptions: **java.lang.RuntimeException:Failed to get driver instance for jdbcUrl=jdbc:mariadb://localhost:3306/jhipster** or **java.lang.RuntimeException: Failed to get driver instance for jdbcUrl=jdbc:mariadb://mariadb:3306/jhipster**

We checked the same variant using Maven as build tool instead of Gradle and it passed both the compilation and the build. The fault is then depending on the build tool rather than the database itself.

This issue was corrected by "ruddel" in pull request 4222[2], by updating *generators/server/templates/gradle/_liquibase.gradle* template file. Since then we have tested the configuration on a more recent version of JHipster and the configuration ran well.

F.3. UAA authentication with Docker

This fault concerns the use of UAA as authentication type while building with Docker. This authentication mechanism is only available in Microservice applications and Microservice Gateways. It is also the default (and only) option for UAA server (UAA as application type), but the it doesn't seem to affect these variants.

This issue occurs when we try to deploy the application (the Microservice) with Docker. When executing the *docker-compose* command, the console will prompt a stacktrace in which we can find *nested exception is java.lang.IllegalStateException: No instances available for uaa.*

When deploying a JHipster app with Docker, and as mentioned on JHipster official website, Docker will launch everything needed (database service, JHipster Registry for microservices, etc.). The cause of this issue is that there are currently no ready-to-use UAA Docker image leaving Docker unable to start the server. In its defence, the JHipster team does notify the user (during the configuration process and on their website) that UAA is still in "Beta" and to *"Use it at your own risk!"*.

To the best of our knowledge this issue still occurs in more recent JHipster version since no UAA Docker image is available.

[2] `https://github.com/jhipster/generator-jhipster/pull/4222`

F.4. UAA authentication with Ehcache as Hibernate 2nd level cache

Some variants (Microservice applications and Microservice Gateways) using an UAA server as authentication also experience another issue.

When building the variant "manually" (with Maven or Gradle), the variant is unable to reach the deployed UAA server. This fault seems to be directly linked to the Hibernate 2nd level cache value. Listing F.3 and Listing F.4 show, respectively, a configuration in which this fault occurs and a configuration without fault. We can see in these listing that the two configurations are the same except one uses EhCache and the other HazelCast as Hibernate 2nd level cache.

We weren't the only one to experience this fault, as attests this issue: `https://github.com/jhipster/generator-jhipster/issues/4005`. It also seems to have been solved in commits on September 28, 2016. The changes can be found here: `https://github.com/jhipster/generator-jhipster/pull/4225/commits`.

Listing F.3: yo-rc.json of a failing variant

```
{
  "generator-jhipster": {
    "jhipsterVersion": "3.6.1",
    "baseName": "gateway2",
    "packageName":
        "io.variability.jhipster",
    "packageFolder":
        "io/variability/jhipster",
    "serverPort": "8080",
    "authenticationType": "uaa",
    "uaaBaseName": "uaa",
    "hibernateCache": "ehcache",
    "clusteredHttpSession":
        "hazelcast",
    "websocket": "no",
    "databaseType": "sql",
    "devDatabaseType": "mysql",
    "prodDatabaseType": "mysql",
    "searchEngine": "no",
    "buildTool": "gradle",
    "useSass": true,
    "applicationType": "gateway",
    "testFrameworks": [
      "gatling",
      "cucumber",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": false
  }
}
```

Listing F.4: yo-rc.json of a working variant

```
{
  "generator-jhipster": {
    "jhipsterVersion": "3.6.1",
    "baseName": "gateway2",
    "packageName":
        "io.variability.jhipster",
    "packageFolder":
        "io/variability/jhipster",
    "serverPort": "8080",
    "authenticationType": "uaa",
    "uaaBaseName": "uaa",
    "hibernateCache": "hazelcast",
    "clusteredHttpSession":
        "hazelcast",
    "websocket": "no",
    "databaseType": "sql",
    "devDatabaseType": "mysql",
    "prodDatabaseType": "mysql",
    "searchEngine": "no",
    "buildTool": "gradle",
    "useSass": true,
    "applicationType": "gateway",
    "testFrameworks": [
      "gatling",
      "cucumber",
      "protractor"
    ],
    "jhiPrefix": "jhi",
    "enableTranslation": false
  }
}
```

F.5. OAuth2 authentication with SQL database

This issue is faced when trying to deploy an application using a SQL database (with Mysql, PostgreSQL or MariaDB) and an OAuth2 authentication with Docker. It manifests itself with the following exception in the output log:
No qualifying bean of type [org.springframework.security.oauth2.provider.token.store.JdbcTokenStore] found for dependency [org.springframework.security.oauth2.provider.token.store.JdbcTokenStore]: expected at least 1 bean which qualifies as autowire candidate for this dependency. Dependency annotations: @javax.inject.Inject()

This problem was reported by "*tsaqova*" in issue #4009 (https://github.com/jhipster/generator-jhipster/issues/4009). The solution proposed by the poster is to modify the file *"/generators/server/templates/src/main/java/package/config/_OAuth2ServerConfiguration.java"* to add the following bean declaration:

```
@Bean
public JdbcTokenStore jdbcTokenStore() {
        return (JdbcTokenStore) tokenStore();
 }
```

As mentioned in the post, JHipster's team was unable to reproduce the error and so to find the source of this issue. We do not know either the root cause, but we still experienced the fault in version 3.9.1 of JHipster. The proposed solution, however, seemed to indeed fix the problem.

F.6. Social Login with MongoDB

This issue occurs in MongoDB variants with social sign-in enabled. When we try to compile such variants, we get the following error message: "*SocialUserConnection.java:3: error: package org.hibernate.annotations does not exist import org.hibernate.annotations.Cache;*.

The issue lies within a template file: *generators/server/templates/src/main/java/package/domain/_SocialUserConnection.java*. The import of hibernate package isn't under conditional pre-processing but it should. Indeed, when scaffolding a JHipster app with MongoDB, the dependency for hibernate annotations is not added, thus leading to a compile exception. This missing dependency occurs both with Maven and Gradle.

It has been fixed in version 3.7.0 of JHipster, by *"ruddell"*, in pull request 4037[3].

[3] `https://github.com/jhipster/generator-jhipster/pull/4037`

G. Failures per individual feature

In section 6.1 we present a Figure 6.1 that summarizes the proportion of failures per individual feature. For the sake of conciseness, we focus on a feature subset of the feature model presented in Figure 4.3).

In this Appendix, we present the others proportions of failures per individual feature not presented in Figure 6.1. We only show the database type, the use of Docker and the choice of the build tool as other features because other features, for instance *Internationalization* or *Hibernate Cache*, show approximately a same ratio of failures per feature. Therefore, we are not able to conclude anything on these results for these features.

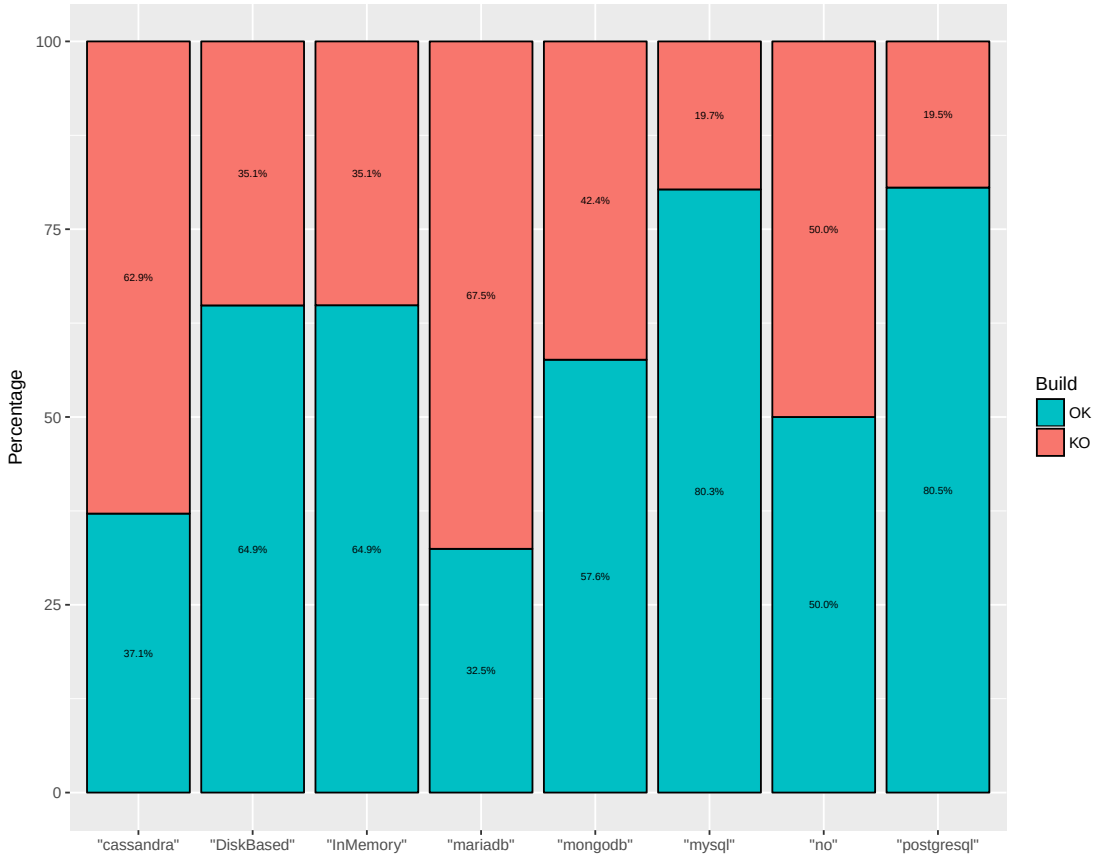

Figure G.1.: Proportion of build failure by database type

G.1. Database type

This difference among features is also noticeable in the database selection: *MariaDB*, *Cassandra* and *MongoDB* related variants show respectively **67.18**% (5708 web-apps), **62.85**% (174 configs) and **42.37**% (200 configs), while *MySQL-based* and *PostgreSQL-based* web-applications have an inferior failing percentage of **19.54**% (1660 configs) and **19.11**% (1624 configs). Figure G.1 shows failures for production databases but also presents results for development databases: *MariaDB*, *Cassandra*, *MongoDB*, *MySQL* and *PostgreSQL* related variants have the same percentage of failures from production databases. *DiskBased* and *InMemory* are special H2-based variants if the developer choose the development database profile, with respectively and approximately the same percentage of failures: **35.15**% (*2986*) and **35.12**% (*2984*) failing configurations. Specific micro-service-based configurations (16) allow to the user to not have database, with a failing percentage of **50**% (8 configurations).

G.2. Docker

Regarding the figure G.2 we observe that there are more failures web-applications when Docker is used than when it is not. There are **41.9**% (*5485*) failing configurations when we use Docker and **29.6**% (*3891*) failing configurations when we don't use Docker, but a standard build tool *Maven* or *Gradle*.

G.3. BuildTool

This disparity is also found in the choice of the build tool. *Gadle* and *HTTP Sessions* suffers from **46.5**% (*6105*) failing configurations, more than *Maven* with respectively **24.91**% (*3271*) failing configurations.

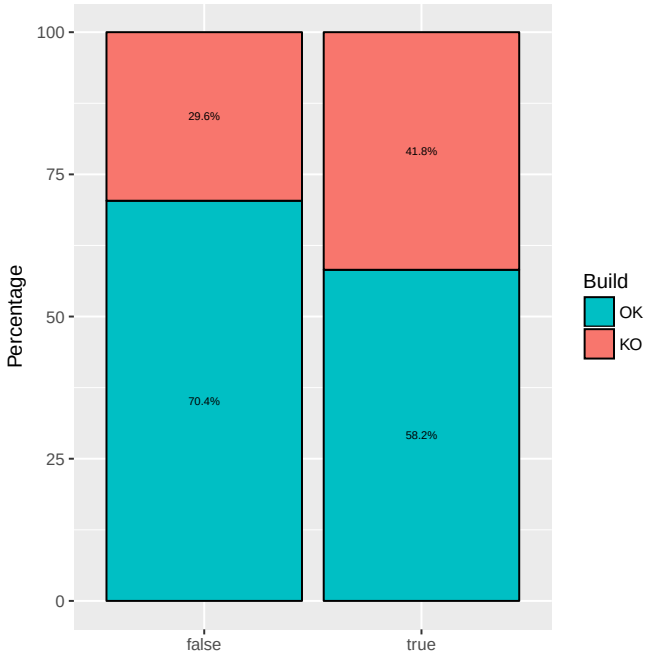

Figure G.2.: Proportion of build failure using docker

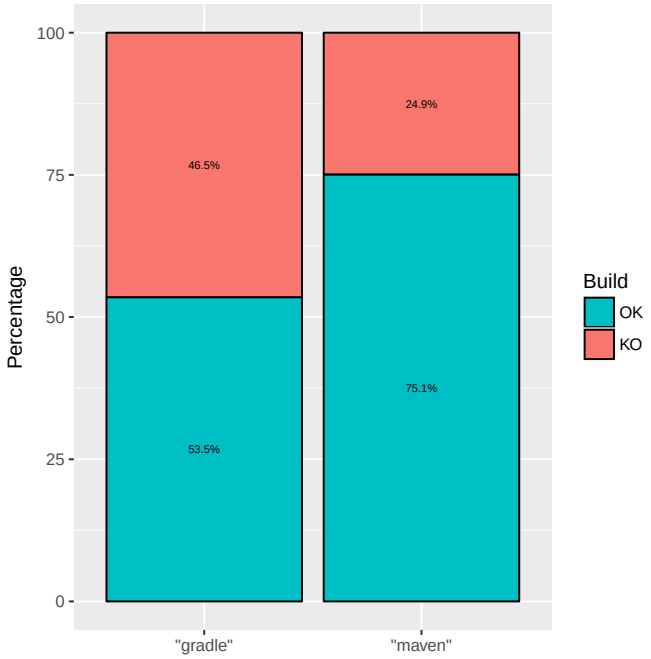

Figure G.3.: Proportion of build failure by build tool type

H. VaMoS'17

In the early stages of our research internship, we also participated with our supervisors and research mentor to the drafting of a scientific article which aimed to introduce our work to the research community.

At the time of writing this article, our derivation and testing infrastructure was almost complete (it was updated later on to mitigate threats such as the presence of false-positives) and we already had some preliminary results (the workflow was ran across $\approx$ 300 configurations).

The article was submitted, and accepted, at the *11th International Workshop on Variability Modelling of Software-intensive Systems*. We thus went to Eindhoven with Gilles Perrouin and Xavier Devroey to present it to the research community. It was overall well received and it allowed us to get some feedbacks which lead us to improve and continue the study.

We present here-after the paper as it was submitted and published.

Yo Variability!
JHipster: A Playground for Web-Apps Analyses

Axel Halin
PReCISE Research Center
University of Namur, Belgium
axel.halin@student.unamur.be

Alexandre Nuttinck
PReCISE Research Center
University of Namur, Belgium
alexandre.nuttinck@student.unamur.be

Mathieu Acher
IRISA, University of Rennes I, France
mathieu.acher@irisa.fr

Xavier Devroey
PReCISE Research Center
University of Namur, Belgium
xavier.devroey@unamur.be

Gilles Perrouin
PReCISE Research Center
University of Namur, Belgium
gilles.perrouin@unamur.be

Patrick Heymans
PReCISE Research Center
University of Namur, Belgium
patrick.heymans@unamur.be

ABSTRACT

Though variability is everywhere, there has always been a shortage of publicly available cases for assessing variability-aware tools and techniques as well as supports for teaching variability-related concepts. Historical software product lines contains industrial secrets their owners do not want to disclose to a wide audience. The open source community contributed to large-scale cases such as Eclipse, Linux kernels, or web-based plugin systems (Drupal, WordPress). To assess accuracy of sampling and prediction approaches (bugs, performance), a case where all products can be enumerated is desirable. As configuration issues do not lie within only one place but are scattered across technologies and assets, a case exposing such diversity is an additional asset. To this end, we present in this paper our efforts in building an explicit product line on top of JHipster, an industrial open-source Web-app configurator that is both manageable in terms of configurations ($\approx$ 163,000) and diverse in terms of technologies used. We present our efforts in building a variability-aware chain on top of JHipster's configurator and lessons learned using it as a teaching case at the University of Rennes. We also sketch the diversity of analyses that can be performed with our infrastructure as well as early issues found using it. Our long term goal is both to support students and researchers studying variability analysis and JHipster developers in the maintenance and evolution of their tools.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging; Empirical software validation;** *Software configuration management and version control systems; Software product lines;* •**Social and professional topics** → **Software engineering education;**

VaMoS '17, February 01-03, 2017, Eindhoven, Netherlands

DOI: http://dx.doi.org/10.1145/3023956.3023963

Keywords

Case Study; Web-apps; Variability-related Analyses

1. INTRODUCTION

JHipster [20] is an open-source generator for Web applications (Web-apps). Started in 2013 by Julien Dubois, JHipster aims at supporting all cumbersome aspects of Web applications development: choice of technologies on the client and server sides as well as integrating them in a complete building process. On the server side, JHipster relies on a Java stack (with Spring Boot). On the client side AngularJS and Bootstrap (a HTML/CSS and JavaScript framework) are used. Finally, Yeoman, Bower, Gulp and Maven automate the building process, including the management of dependencies across the offered technologies [20]. JHipster is used all over the world both by independent developers and large companies[1] such as Adobe, Google, HBO, *etc.*.

The setup of a Web-app with JHipster is performed in two phases: *configuration* and *generation*. The configuration is done via a command-line interface (see Figure 1) through which the user can select the technologies that will be included. The result of this configuration is a `yo-rc.json` file (see Listing 1) used for the generation phase. To achieve this generation, JHipster relies on Yeoman[2] using *npm* and *Bower* tools to manage dependencies, and *yo* tool to scaffold projects or useful pieces of an application [41]. Based on the content of the `yo-rc.json` file, JHipster's generator produces relevant artefacts (Java classes and so on). Beyond this generator – the main focus of this paper – JHipster offers multiple sub-generators and even has its own language, *JHipster Domain Language*, to easily generate entities and all related artefacts (e.g., Spring Service Beans).

From its inception to this day, JHipster has constantly grown throughout 146 releases. It now has more than 5000 stars on GitHub and can count on a community of 250 contributors. In October 2016, it has been downloaded 22739 times[3]. This constant evolution allows JHipster to offer up-to-date frameworks and technologies to its users (for instance, the infrastructure can be generated using Docker since release 3.0.0).

[1]https://jhipster.github.io/companies-using-jhipster/
[2]http://yeoman.io/
[3]https://www.npmjs.com/package/generator-jhipster

Listing 1: _yo-rc.json excerpt

```
{
  "generator-jhipster": {
    (...)
    "useSass": false,
    "applicationType": "monolith",
    "testFrameworks": [],
    "jhiPrefix": "jhi",
    "enableTranslation": false
  }
}
```

Figure 1: JHipster command line interface

Listing 2: server/prompt.js excerpt

```
(...)
when: function (response) {
    return applicationType === 'microservice';
  },
  type: 'list',
  name: 'databaseType',
  message: function (response) {
      return getNumberedQuestion('Which *type* of
          database would you like to use?',
          applicationType === 'microservice');},
  choices: [
      {value: 'no', name: 'No database'},
      {value: 'sql', name: 'SQL (H2, MySQL, MariaDB,
           PostgreSQL, Oracle)'},
      {value: 'mongodb', name: 'MongoDB'},
      {value: 'cassandra',name: 'Cassandra'}
  ],
  default: 1
(...)
```

By combining configuration and generation in a constantly evolving stack of technologies, JHipster is akin to Mr Jourdain's prose: a software product line initiative without naming it as such. In this paper, we describe our efforts in building an explicit product line on top of JHipster to expose it as a case for research, education and to ease the development of JHipster itself. Our preliminary infrastructure applied on only 300 variants (out of $\approx$ 160,000) already disclosed some unreported issues, which we perceive as an incentive to pursue in this direction. Though "lifting" such infrastructure in the web domain is not new (*e.g.*, [43, 36]), JHipster offers interesting assets beyond replication studies: *(a)* it covers key aspects of product line development, variability, product derivation and evolution; *(b)* the number of variants is large enough to require automated derivation support (on top of Yeoman) but small enough to be enumerated through distributed computing facilities yielding exact results to assess various kinds of analyses; *(c)* it allows to address variability modelling and configuration challenges across technological spaces [21]. All sources of our preliminary infrastructure can be found at https://github.com/axel-halin/Thesis-JHipster.

In the remainder, we present our efforts to manually reverse engineer variability from JHipster artefacts and define a Web-app software product line (Section 2). We analyse current state of the art for products' analyses, family-based analyses, and product line evolution in Section 3, and presents the JHipster's potential for each of those research fields. Section 4 reports our experiences in using JHipster as an education case study for software product line teaching. Finally, Section 5 concludes this paper and presents future works using JHipster.

2. JHIPSTER AS A PRODUCT LINE

Although never explicitly acknowledged by the JHipster developer, it is straightforward to think JHipster supported technologies[4] (microservice architecture, authentication, *etc.*) as variation points to be resolved during product line application engineering.

Based on this vision, we decided to model the system in a feature model using FAMILIAR[1]. This decision was motivated by our will to assess automatically a maximum of configurations authorized by the JHipster generators. The first step was to identify the variability. To do so, we retrieved the publicly available source code[5] and analysed it. We quickly identified interesting artefacts: `prompts.js` files. JHipster's Yeoman generator is divided in multiple `prompts.js` files, each of which handles specific parts of the configuration process. For instance, `client/prompts.js` offers the possibility to use *LibSass*, while, as illustrated in Listing 2, the type of database is selected in `server/prompts.js`.

From these artefacts, we derived the feature model presented in Figure 2. In this model, the abstract features represent the multiple choices questions (typically, which of these technologies do you wish to use?) while the concrete features are the different choice(s) available to the user. Except for the testing frameworks, all of these multiple-choice questions are exclusive (choose only one production database, for instance), mapped as alternate groups. Yes or no questions are represented by optional features. So, if we consider Listing 2 as an example we have: *database* as an optional abstract feature, with *SQL*, *Mongodb* and *Cassandra* as concrete alternate sub-features. We also identified several constraints in the JavaScript files (`when (...) return applicationType === 'microservice'`, in Listing 2, is one of them) which we synthesized in 15 constraints. For the sake of conciseness, we only present few of them in Figure 2.

At the variability realization level, JHipster relies on Yeoman template files (JavaScript, Java, HTML, XML, ...) for holding common parts but also properties specific to some variants. Conditional compilation is the main implementation mechanism for realizing variability. With Yeoman templates, some specific code in the different artefacts is activated depending on user's configuration. A first example is given in Listing 3 for Maven files: `hikaricp.version` Maven property is defined only if the configuration includes an SQL database. A second example in given in Listing 4 for Java files. The method *h2TCPServer* is only used in configurations relying on H2 databases (either h2Disk or h2Memory). The inheritance of AbstractMongoConfiguration depends on the activation of *mongodb*. Java annotations are also subject to variations. Thus, variability information is scattered in

[4]The complete list is available on https://jhipster.github.io/

[5]For this study, we use JHipster v3.6.1: https://github.com/jhipster/generator-jhipster/releases/tag/v3.6.1

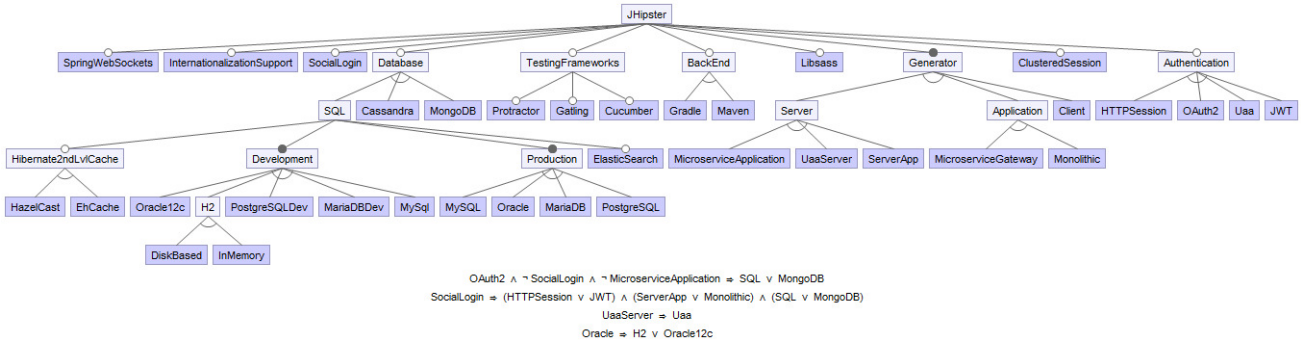

Figure 2: JHipster reverse engineered feature model

Listing 3: `_pom.xml` excerpt

```
<?xml version="1.0" encoding="UTF-8"?>
(...)
  <properties>
      <%_ if (databaseType === 'sql') { _%>
      <hikaricp.version>2.4.6</hikaricp.version>
      <%_ } _%>
      <awaitility.version>1.7.0</awaitility.version>
      (...)
  <properties>
(...)
```

Listing 4: `_DatabaseConfiguration.java` Excerpt

```
(...)
@Configuration<% if (databaseType == 'sql') { %>
@EnableJpaRepositories("<%=packageName%>.repository")
@EnableJpaAuditing(...)
@EnableTransactionManagement<% } %>
(...)
public class DatabaseConfiguration
<% if (databaseType == 'mongodb') { %>
        extends AbstractMongoConfiguration
<% } %>{

 <%_ if (devDatabaseType == 'h2Disk' ||
      devDatabaseType == 'h2Memory') { _%>
   /**
   * Open the TCP port for the H2 database.
   * @return the H2 database TCP server
   * @throws SQLException if the server failed to
       start
   */
   @Bean(initMethod = "start", destroyMethod =
       "stop")
   @Profile(Constants.SPRING_PROFILE_DEVELOPMENT)
   public Server h2TCPServer() throws SQLException {
      return Server.createTcpServer(...);
   }
  <%_ } _%>
(...)
```

different artefacts (e.g., Maven, Java, JavaScript, *etc.*).

2.1 Analysis Workflow

From the feature model, we devised an automated way to generate JHipster's variants in order to check their validity (are the variants correctly generated? do the Web-apps compile? Can we build them?). This was done by building for each variant the matching `.yo-rc.json` file and then calling the generator (*yo jhipster*) on each of them. The keys Yeoman expects to find in the `.yo-rc.json` file can be found in the function *saveConfig* of JHipster's `index.js` files. From there we can then run variant dependent commands (Maven or Gradle, Docker, ...) to compile, build and test them (see Figure 4). For each variant, we store some interesting information in a CSV file to analyse later on. Currently, this information is the result of the generation/-compilation/build processes; the logs from each process if there is a problem; the duration of the generation, compilation, and build phases; the size of the Docker image; and the results from the unit tests. This analysis workflow currently runs on a subset of JHipster's variants: we do not, yet, generate client or server standalone application variants (which may be obtained using JHipster sub-generator); we exclude variants with an external databases (Oracle or H2); and we include all test frameworks in each variant (if the constraints allow it), as it prevents the generation of similar variants with only different test frameworks. These choices were motivated both by technical reasons (Oracle being a proprietary database additional work is needed to use it properly) and practical reasons (What can we test with client only app? Are client/server parts not tested in the other types of application?). This selection decreased the total number of variants to about 4600.

2.2 Preliminary Analyses

With the analysis workflow presented in Figure 4, we have already tested the validity (generation, compilation, build) of about 300 variants and found unreported bugs (*i.e.*, anything that would prevent the generation, compilation, or execution of a variant of JHipster, or lead to deviant behaviour) in a few of them. For example, the first bug we found is related to the Docker image of the MariaDB database, in monolithic applications, and it is encountered while trying to deploy the application via Docker. Basically, Docker is looking for the wrong repository/tag, one that doesn't exist. The cause of this error is a missing line in the file `/src/main/docker/app.yml`: a condition *prodDatabaseType == 'mariadb'* on line 5. This issue is still found in JHipster 3.9.1 and doesn't seem to have been detected (currently no related issue post mention it). We will extend the number of tested variants to possibly all of them. We will also use the 3 supported testing frameworks (Cucumber, Protractor and Gatling) to evaluate beyond the correctness of the applications their non-functional properties (perfor-

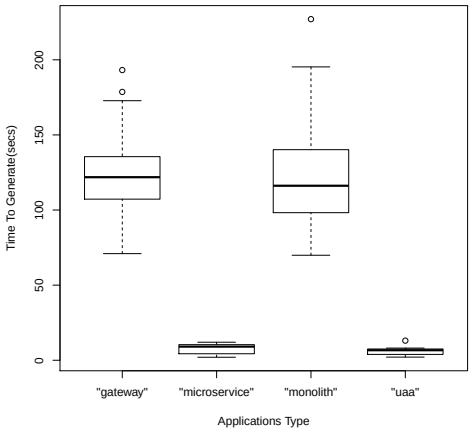

Figure 3: Distribution of generation time by application type boxplot

mance testing, UI testing, *etc.* see Section 3).

We started investigating preliminary results. For instance, the correlation between generation time and the type of the application (see Figure 3). We observe on this boxplot that micro-service applications and UAA servers require shorter generation times than monolithic applications or micro-service gateways. Indeed, micro-service applications and UAA servers do not need the client part of JHipster's applications. We hope to extract information regarding non-functional properties of the generated Web-apps.

3. A CASE FOR RESEARCHERS

In the previous section, we presented our derivation infrastructure as well as a few statistics on the generated products and issues found. In this section, we explore two vertices of the "PLA cube" [51]: product-based and family-based analyses. We explain why JHipster is a good candidate to devise new techniques and perform additional empirical assessment of existing ones.

3.1 Products' Analyses

Product lines usually allow a large number of products. Two approaches are possible to validate them. The use of formal methods which prove correctness properties in the specification at the product line level such that all derived products satisfy the same properties, without needing to enumerate all of them [49, 5]. Another approach is to rely on testing, which main goal is to select and sort the fittest set of products to test according to given criteria in order to detect as much bugs as possible. Systematic studies show that a lot of effort has been put on SPL testing [8, 13, 32].

3.1.1 Structural Sampling

Sampling techniques. To reduce the number of products to test, one popular research direction is to use Combinatorial Interaction Testing (CIT) techniques [6, 31] and pairwise (generalized to t-wise) criteria [28, 30, 33, 40]. Over the years, several tools have been developed and support pairwise based selection on the feature model, e.g., [18, 22]. In order to support larger t values, as well as larger feature models, other search-based heuristics have been proposed [3, 17, 45, 37]. All of those CIT, t-wise, and other search-based techniques make the hypothesis that bugs come from interactions between few features and try to select an adequate set of products to test in order to cover as much feature combinations as possible. They have been extensively validated on a large number of feature models, with different sizes, and coming from different sources. However, very few evaluations have actually built the set of products to test in their process.

JHipster potential for Sampling. With $\approx$ 163,000 possible products, JHipster is both non-trivial (as opposed to some academic models in the SPLOT repository) without being as large a Linux, WordPress or Drupal cases. This particularity makes accessible the generation of ***all*** the variants. The idea is to be able to obtain a *ground truth* to compare the efficiency of sampling algorithms. By being able to compute absolute values for the numbers and types of interaction bugs, biases when assessing techniques can be reduced. As noted by Jin *et al.* [21], configuration issues can happen everywhere: we believe that the variety of technologies at work in a JHipster derived product is also an opportunity to study such aspects. As seen in Section 2, our feature model integrates variability information from different files and will lead researchers to study different kinds of interaction bugs.

3.1.2 Functional Testing

Product-level functional testing. As noted by Von Rhein *et al.* [51], product-based analysis strategy is simple: we analyse each product individually without taking into account variability (it has been resolved using sampling or enumerating all products). The benefit is that single-product analysis tools can be used. Researchers have proposed to derive test cases from product line scenarios and use cases (e.g., [34]) promoting the reuse of test models and artefacts.

JHipster's potential. As opposed to Drupal or Wordpress cases, where test cases are either optional or solely depending on the will of plugin developers [43, 36, 15], JHipster comes with a systematic testing infrastructure and test cases are deployed for every Web-app deployed. In particular, Cucumber [7] supports early testing in the form of scenarios. Integration with code coverage tools is also available. However our preliminary analyses shown that the provided tests were quite simple, product-agnostic (based on a generic application that is the root of all products) and code coverage was quite low. Thus, we should derive test cases that take into account the specificities of each product, to get a better *base coverage* prior to the development of a richer Web-app on top of the derived product.

3.1.3 Non-Functional Analyses

Feature-related quality attributes. Recent research shifts from functional validation using testing to detect undesired feature interactions to non-functional analyses in order to predict performance of a given product [44, 47, 48, 46]. Using statistical learning [16] and regression methods [50], or mathematical models to predict and detect (undesired) performance-relevant feature interactions [53].

JHipster potential for quality analyses. Web-apps are particularly interesting cases for performance, since this quality attribute has a direct influence on Websites' suc-

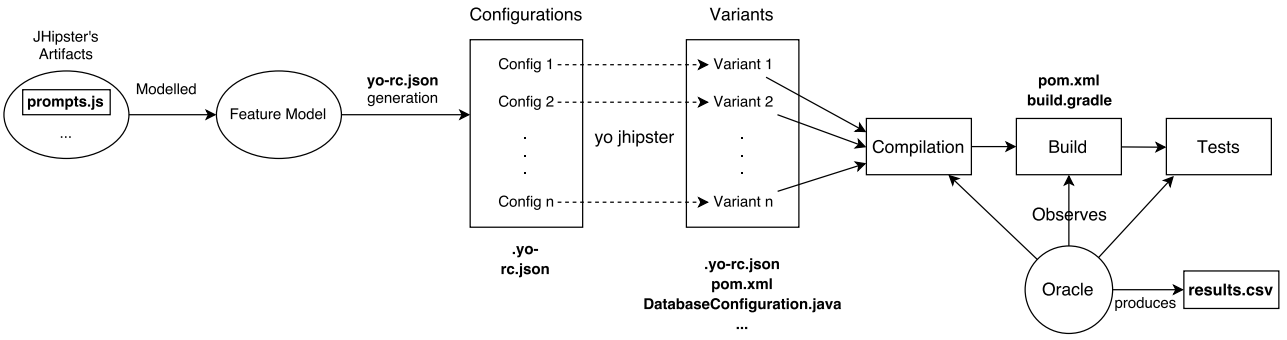

Figure 4: Complete Analysis Workflow

cesses. To this end, JHipster comes with Gatling[6], a load testing tool. It is possible to experiment with feature-related performance techniques in order to assess the proposed theories and calibrate statistical learning. Note that performance is not the only quality attribute that can be studied: security is also key especially for e-commerce websites. While the JHipster infrastructure does not currently offer any security-dedicated analysis toolset, the diversity of technologies used in a JHipster application and our automated derivation approach allow to focus on a given technology in various security scenarios.

3.2 Family-based Analyses

Variability-aware techniques. So far, our infrastructure focuses on the product level by sampling products of interest and analysing them individually using provided validation environments (Protactor, Karma.js, Cucumber, etc.). While our goal is to obtain a ground truth by analysing all the variants [14], exploiting variability to reuse analyses (e.g. tests, proofs, etc.) in order to reduce the overall analysis effort and better scale large cases is relevant [27]. Model-based testing approaches use behavioural models of the product line to generate test cases for the different products: (resp.) delta-oriented product line testing [29, 26] and featured transition system based [10, 9, 11] approaches use (resp.) state machines and transition systems in order to capture the common and product specific behaviour of the product line. At the code level, variability-aware parsers [24], variational structures [52] and type-checking [23] are of interest. They enable *variability-aware testing* [25] to, for example, evaluate a test case against myriads of configurations in one run [36].

JHipster potential. JHipster offers an interesting playground for the aforementioned analyses. The difficulty for model-based approaches is to obtain accurate models of the case study when the implementation already exists. To this end, server execution logs of different variants can be used to extract part of the system behaviour [9]. At the code level, the challenge is to specify annotations across multiple technological spaces, while performing commonality and variability analysis. JHipster provides a generic customisable Web-app for each variant, enabling a user to log in and to create entities, as a starting point. JHipster also includes a configurator, allowing to consider the interaction between configuration workflows [19] and derived products [39].

[6]http://gatling.io/

3.3 Product Line Evolution

Evolution techniques. From the evolution perspective, product lines represent an interesting challenge. Product lines developers have to manage updates at different levels: the evolution of the variability model and the mapping to other artefacts [12]; the evolution of the artefacts themselves which will impact several products [35, 42]; and the evolution of the configurator and configuration workflow. To understand how existing SPL are updated, Passos *et al.* [38] recently studied the Linux kernel variability models and other artefact types co-evolution.

JHipster potential. With 146 releases since 2013, JHipster is under active development and evolution. Therefore a challenge for researchers is to devise automated means to update the JHipster feature model. As opposed to the Linux case, where part of the variability model can be extracted from KConfig, several JavaScript files are necessary to build it, pushing for more versatile variability inference techniques.

4. A CASE FOR EDUCATION

JHipster has been used as part of different teaching courses. In this section, we report on such an experience and then argue that JHipster is a relevant case for education and in particular for SPL teaching.

4.1 Experiences

Experience #1. Our first teaching experience with JHipster started in 2015 at University of Rennes 1. The audience was 40+ MSc students with a speciality in software engineering or in software management. As part of a model-driven engineering course, we used to teach variability modelling and implementation techniques. In 2015, we decided to slightly change the way variability is explained and we notably introduced JHipster, for the following reasons.

First, students used JHipster in another course dedicated to Web development. Therefore students could reuse JHipster for building a quite complex Web application in the model-driven engineering course: a Web generator of video variants called *VideoGen*. *VideoGen* is a software application that builds video variants by assembling different video sequences; it is a generalization of a real-world Web generator [4]. Video variants can be randomly chosen or users can configure their videos through a Web interface. A textual

specification, written in a domain-specific language, documents what video sequences are mandatory, optional, or alternatives. Frequencies and constraints can also be specified[7]. *VideoGen* challenges students to master Web development as well as variability modelling and implementation techniques: they should build a Web configurator, implement algorithms for randomly choosing and building a video variant, etc. The video generator was the running example of the course and was used in the lab sessions and in the project for evaluating students. JHipster was used all along to implement the Web application, including a Web configurator. In summary, JHipster was used as a relevant technology for showing the relations with other courses (Web development) and for implementing a non-trivial variability system (a video generator) based on modelling technologies.

The second reason is that we took the opportunity to explain *how* JHipster is implemented and more precisely how variability concepts and techniques are applied in practice. During the course, we used JHipster to define what a software product line is, making the correspondences with other well-known configurable systems like Linux, Firefox, or ffmpeg. We explained variability implementation techniques and in particular conditional compilation, templates and annotative-based approaches with the use of JHipster. From a variability modelling perspective, we introduced feature models by using the configurator of JHipster. In the lab sessions, students used the JHipster generator to obtain a Web stack and develop the video generator. They had to make the Web video generator configurable, for instance, they had to implement the ability to save or not a video variant. We have also proposed different exercises related to feature modelling. Along the way, students could exercise on variability concepts that were also found in JHipster.

Our experience was mostly positive. The evaluation of the students' projects on the Web video generator gives high marks. Interactions with students during the courses show that JHipster helps to understand more concretely variability concepts. However we noticed two limitations. First, students manipulated variability concepts at two levels and for two different purposes. The first level was for creating from scratch a configurable video generator, involving skills in domain-specific languages, model transformations, and variability modeling. The second level was for understanding and reusing the JHipster generator. There was some confusions between the two levels. The explanations on JHipster certainly deserve more time and a specific attention – perhaps a dedicated exercise, see hereafter. Second, the technology behind JHipster is quite advanced and requires numerous skills. Some students have technical difficulties to connect the dots and transfer their conceptual knowledge into concrete terms. We had to postpone the deadline for project delivery to let students enough time to master the Web stacks.

For mitigating the two weaknesses, we have decided in 2016 to play the full course on Web development *before* the model-driven engineering and variability courses. We expect that students can, prior to the course, master JHipster for (1) better understanding its internals; (2) better implementing the variability concepts.

Experience #2. Our second teaching experience with JHipster was in late 2015 at University of Rennes 1. This time the audience was MSc students with a strong interest in research. We reused almost the same material as previously but we also addressed more advanced topics like automated reasoning with solvers, software product line verification and validation, *etc.*. We used JHipster for the same previous reasons. Compared to the first experience, JHipster was the sole focus of this course and there was no video generator to develop. It simplified how variability concepts were introduced and explained. The project aimed at evaluating students and was oriented for addressing some open research questions: How to elaborate and reverse engineer a feature model of the JHipster generator? What are the configuration bugs of JHipster? How to automatically find those bugs?

With the JHipster case, students could elaborate a feature model based on a static analysis of several artefacts. They could apprehend the combinatorial explosion inherent to variability-intensive systems. Overall students could revisit the variability techniques of the course with a realistic and complex example. The JHipster case also shows to students the connection with other research works, mainly what we have described in the previous section. Some questions were voluntary open like the proposal of "a strategy for testing the configurations of JHipster at each commit or release".

Another motivation for us was to use JHipster to explore some research directions and make some progress with students. We asked them to collect and classify configuration bugs on GitHub. We also gathered several feature models based on their analysis. Such works help us to re-engineer JHipster as a software product line: we reused such feature models to initiate the work exposed in Section 2. Students of the course did not design or develop the workflow analysis of Figure 4. Discussions and insights, however, motivate the need to build such a testing infrastructure for JHipster.

Overall, the work of students was evaluated in a positive way. They demonstrated their abilities to understand and use variability concepts. It was also useful for our own research work.

Other experiences. We have used JHipster in other educating settings in 2015: *(a)* at University of Montpellier for MSc students; *(b)* within the DiverSE Inria team for forming PhD students to variability. Such experiences deserve less comments since the duration of the courses was one full-day. Yet the JHipster case was again useful to us, educators and researchers, to both illustrate the variability concepts and exchange on open issues.

4.2 JHipster for Education

A survey on teaching of software product lines showed that two recurring issues for educators are the absence of case studies and the difficulty to integrate product lines within a curriculum [2]. Similar concerns have been raised at SPLTea'14 and SPLTea'15 workshops (see http://spltea.irisa.fr). JHipster acts as an interesting and useful case for addressing these issues. Specifically, JHipster can be used for: (i) illustrating a product line course and for describing variability modelling and implementation techniques with a real-world case over different technologies; (ii) conducting lab sessions in relation with variability; (iii) connecting or better integrating product line courses to other courses (*e.g.*, Web development, model-driven engineering); (iv) exploring

[7]More details can be found online: https://github.com/FAMILIAR-project/teaching/tree/gh-pages/resources/Rennes2015MDECourse

open research directions with students.

In conclusion, our experiences with JHipster were mostly positive, though some improvements can be made. All material (slides, instructions of lab sessions) can be found online: http://teaching.variability.io/. We are reusing the same case and material in 2016 at the University of Rennes 1.

5. CONCLUSION AND FUTURE WORK

In this paper, we described JHipster as a case for experimenting with various kinds of variability-related analyses and teaching software product lines. We introduced an analysis workflow that automates the derivation of JHipster variants (Web-apps) on the basis of a feature model manually extracted from Jhipster questionnaire's files. As the number of possibilities is within reach of current (distributed) computing facilities, some "all-products" information may be obtained, which is useful to assess some specific techniques such as sampling. Our analysis workflow is also relevant for education to understand and explore product line derivation testing and analysis concepts. Our analysis infrastructure is only in its premises and naturally calls for future developments. At the research level, we would like of course to share the results obtained on running analyses on the whole product line. This requires running our workflow on distributed infrastructure like Grid5000 (https://www.grid5000.fr/), an option that we are currently studying. We also want to share our results (bugs, performance issues) with the JHipster developers so that they can take advantage of them in their fixes and releases. We finally would like to introduce this workflow in our SPL teaching curriculum and continue to share it openly with the community.

6. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their feedback. This work was partly supported by the European Commission (FEDER IDEES/CO-INNOVATION).

7. REFERENCES

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[2] M. Acher, R. E. Lopez-Herrejon, and R. Rabiser. A survey on teaching of software product lines. In *VaMoS '14*, Nice, France, jan 2014. ACM.

[3] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: efficient product-line testing using incremental pairwise sampling. In *GPCE '16*, pages 144–155. ACM, 2016.

[4] G. Bécan, M. Acher, J.-M. Jézéquel, and T. Menguy. On the variability secrets of an online video generator. In *Variability Modelling of Software-intensive Systems (VaMoS'15)*, pages 96 – 102, Hildesheim, Germany, jan 2015.

[5] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, aug 2013.

[6] M. Cohen, M. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, 2008.

[7] CucumberTeam. Cucumber website. https://cucumber.io, accessed in November 2016.

[8] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira. A systematic mapping study of software product lines testing. *Information and Software Technology*, 53(5):407–423, 2011.

[9] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. Statistical prioritization for software product line testing: an experience report. *SoSyM*, pages 1–19, 2015.

[10] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-Y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In *ISoLA '14*, volume 8802 of *LNCS*, pages 336–350. Springer, 2014.

[11] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. Search-based Similarity-driven Behavioural SPL Testing. In *VaMoS '16*, pages 89–96. ACM, 2016.

[12] N. Dintzner, A. van Deursen, and M. Pinzger. FEVER: Extracting Feature-oriented Changes from Commits. In *MSR '16*, pages 85–96. ACM, 2016.

[13] E. Engström and P. Runeson. Software product line testing - A systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[14] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides. Exploiting the Enumeration of All Feature Model Configurations. In *SPLC '16*, Beijing, China, Sept. 2016.

[15] M. Greiler, A. van Deursen, and M. A. Storey. Test confessions: A study of testing practices for plug-in systems. In *ICSE '12*, pages 244–254. ACM, 2012.

[16] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware performance prediction: A statistical learning approach. In *ASE '13*, pages 301–311. IEEE, 2013.

[17] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.

[18] A. Hervieu, B. Baudry, and A. Gotlieb. PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models. In *ISSRE '11*, number i, pages 120–129. IEEE, 2011.

[19] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *SPLC '09*, pages 221–230. Carnegie Mellon University, 2009.

[20] JHipsterTeam. Jhipster website. https://jhipster.github.io, accessed in November 2016.

[21] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: implications for testing and debugging in practice. In *ICSE '14 Companion Proceedings*, pages 215–224. ACM, 2014.

[22] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from

large feature models. In *SPLC '12*, volume 1, page 46. ACM, 2012.

[23] C. Kastner and S. Apel. Type-checking software product lines-a formal approach. In *ASE '08*, pages 258–267. IEEE, 2008.

[24] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, volume 46, pages 805–824. ACM, 2011.

[25] C. Kästner, A. Von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8. ACM, 2012.

[26] R. Lachmann, S. Lity, S. Lischke, S. Beddig, S. Schulze, and I. Schaefer. Delta-oriented test case prioritization for integration testing of software product lines. In *SPLC '15*, pages 81–90. ACM, 2015.

[27] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE '13*, pages 81–91. ACM, 2013.

[28] M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, 20(3-4):567–604, 2012.

[29] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In *TAP '12*, volume 7305 of *LNCS*, pages 67–82. Springer, 2012.

[30] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *ICSME '13*, pages 404–407. IEEE, 2013.

[31] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed. A first systematic mapping study on combinatorial interaction testing for software product lines. In *ICSTW '15*, pages 1–10. IEEE, 2015.

[32] I. d. C. Machado, J. D. McGregor, Y. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, 2014.

[33] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical pairwise testing for software product lines. In *SPLC '13*, page 227. ACM, 2013.

[34] C. Nebut, Y. L. Traon, and J. Jézéquel. *System Testing of Product Lines: From Requirements to Test Cases*, pages 447–477. Springer, 2006.

[35] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. Safe evolution templates for software product lines. *Journal of Systems and Software*, 106:42–58, 2015.

[36] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE '14*, pages 907–918. ACM, 2014.

[37] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software*, 122:287–310, 2016.

[38] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, 2016.

[39] G. Perrouin, M. Acher, J.-M. Davril, A. Legay, and P. Heymans. A complexity tale: web configurators. In *VACE '16*, pages 28–31. ACM, 2016.

[40] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2011.

[41] M. Raible. *The JHipster mini-book*. C4Media, 2015.

[42] G. Sampaio, P. Borba, and L. Teixeira. Partially safe evolution of software product lines. In *SPLC '16*, pages 124–133. ACM, 2016.

[43] A. B. Sanchez, S. Segura, and A. Ruiz-Cortes. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *ICST '14*, pages 41–50. IEEE, 2014.

[44] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T). In *ASE '15*, pages 342–352. IEEE, 2015.

[45] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *ICSE '13*, pages 492–501. IEEE, 2013.

[46] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence models for highly configurable systems. In *ESEC/FSE '15*, pages 284–294. ACM, 2015.

[47] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE '12*, pages 167–177. IEEE, 2012.

[48] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[49] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, nov 2015.

[50] P. Valov, J. Guo, and K. Czarnecki. Empirical comparison of regression methods for variability-aware performance prediction. In *SPLC '15*, pages 186–190. ACM, 2015.

[51] A. Von Rhein, S. Apel, C. Kästner, T. Thüm, and I. Schaefer. The pla model: on the combination of product-line analyses. In *VaMoS '13*, page 14. ACM, 2013.

[52] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *SPLASH '14*, pages 213–226. ACM, 2014.

[53] Y. Zhang, J. Guo, E. Blais, K. Czarnecki, and H. Yu. A mathematical model of performance-relevant feature interactions. In *SPLC '16*, pages 25–34. ACM, 2016.